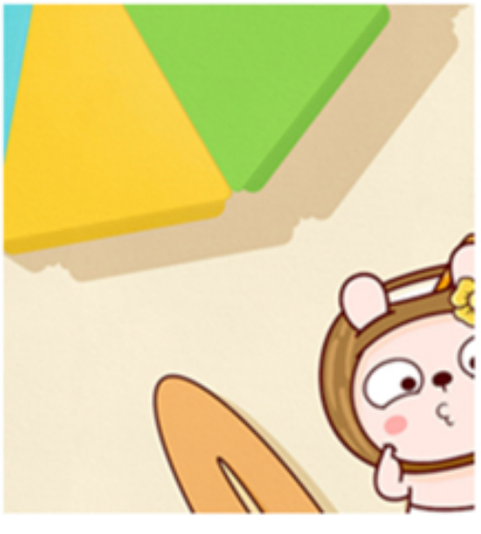
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

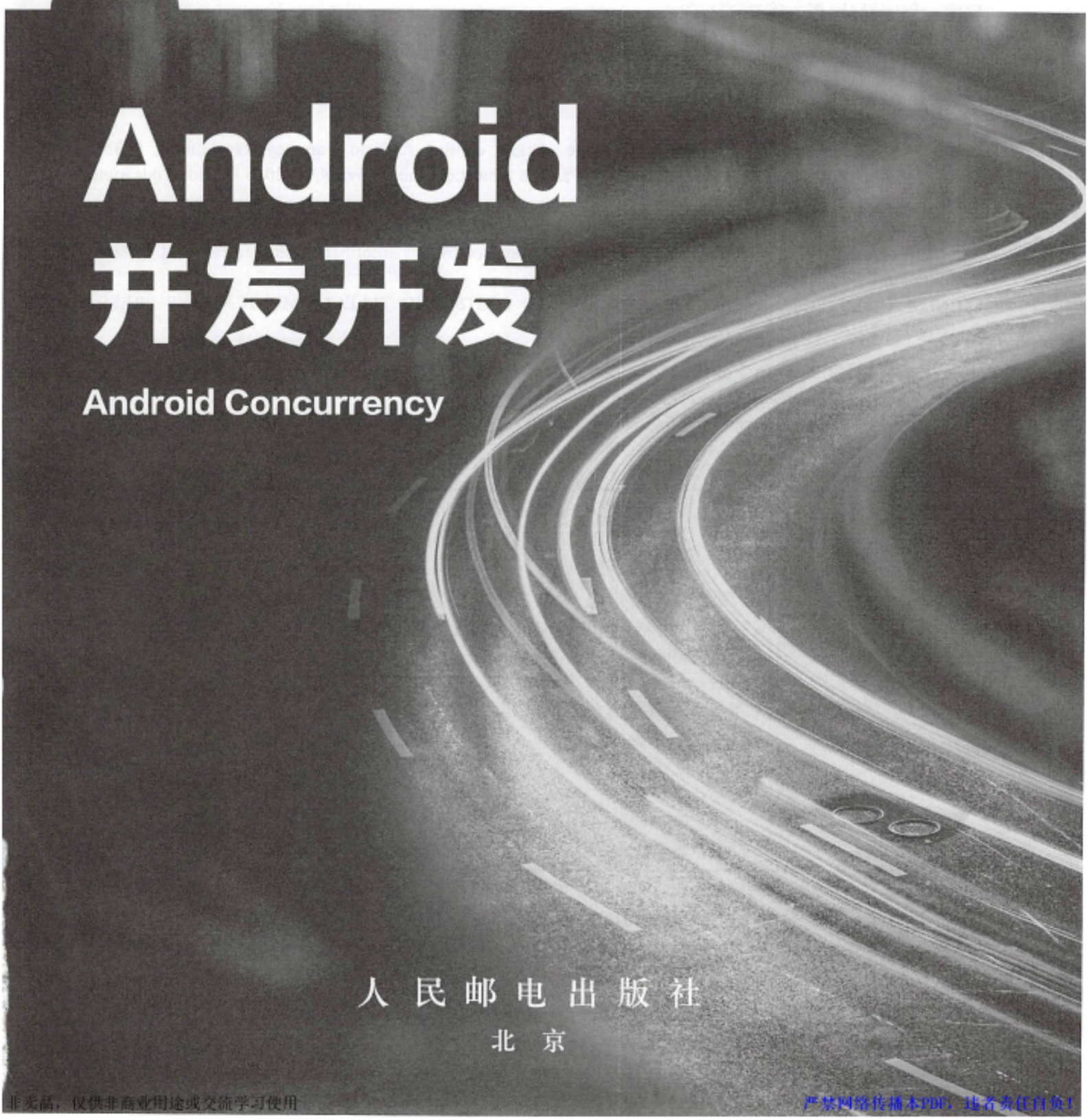
版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！
异步图书 www.epubit.com
非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！
非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！
非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！
非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！
非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

Pearson

[美]G．布莱克·梅克（G．Blake Meike）著
师蓉 译



Android 并发开发

Android Concurrency

人民邮电出版社
北京

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF，违者责任自负！
版权所有，严禁传播，违者自负法律责任！

图书在版编目（CIP）数据

Android并发开发 / (美) G.布莱克·梅克 (G. Blake Meike) 著 ; 师蓉译. -- 北京 : 人民邮电出版社, 2018.10

ISBN 978-7-115-48961-6

Ⅰ. ①A… Ⅱ. ①G… ②师… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2018)第169661号

版权声明

Authorized translation from the English language edition, entitled Android Currency, 1st Edition, ISBN: 0134177436 by MEIKE, G. BLAKE, published by Pearson Education, Inc. Copyright © 2016 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by POSTS AND TELECOMMUNICATIONS PRESS, Copyright © 2018.

本书中文简体版由 **Pearson Education,Inc** 授权人民邮电出版社出版。未经出版者书面许可，不得以任何方式或任何手段复制和抄袭本书内容。

本书封面贴有 **Pearson Education**（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

◆ 著　[美] G. 布莱克·梅克（G. Blake Meike）

译　师　蓉

责任编辑　吴晋瑜

责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号

邮编 100164　电子邮件 315@ptpress.com.cn

网址 http://www.ptpress.com.cn

固安县铭成印刷有限公司印刷

◆ 开本：800×1000　1/16

印张：12.75

字数：278 千字　2018 年 10 月第 1 版

印数：1－2 400 册　2018 年 10 月河北第 1 次印刷

著作权合同登记号　图字：01-2016-7582 号

定价：49.00 元

读者服务热线：(010) 81055410　印装质量热线：(010) 81055316

反盗版热线：(010) 81055315

广告经营许可证：京东工商广登字 20170147 号

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

内容提要

本书共 8 章。第 1 章介绍一个非典型的并发模型，为后文的阐释做好铺垫。第 2 章和第 3 章分别介绍 Java 并发和 Android 应用程序模型，主要介绍 Java 线程、同步、并发包、生命周期和组件、Android 进程等基本概念。第 4 章介绍 AsyncTask 和 Loader。第 5 章～第 7 章是本书的核心内容，深入探讨 Android 操作系统的细节，如 Looper/Handler、Service、Binder、定时任务等。第 8 章介绍并发工具，如静态分析、注解、断言等。

本书适合有一定 Android 开发经验的读者阅读。如果你是一名新手，建议你在掌握相关入门知识的基础上阅读本书，以达到更好的学习效果。本书给出了多段代码，旨在让读者亲自实践后更好地掌握 Android 并发开发的相关内容。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

前 言

我有幸在这个行业工作多年，看到了很多背景下的并发。我在学校读书时，并发只是一个论文主题。作为一名熟练的开发人员，我看到分布式后端系统中的并发代码几乎都是用 Java 编写的。最近，我有机会亲身体验 Erlang 和 Scala 这样的语言，希望能使并发代码的设计和编写更容易。

在我职业生涯早期，一位非常支持我的面试官指导我重新设计了双重检查锁。我还记得自己在一年后发现双重检查锁模式不安全，并在不久后遇到了第一个不正确的字节码时的那种心情。然而，最令人惊讶的也许是我最近从 2015 年编写的代码中删除了双重检查锁模式的那次实践。

在此期间，不变的是这个话题的神秘性和争议。必须使用并发代码时，即便是能够完美胜任工作的老手也会突然犯很幼稚的错误。开发人员有时会就某段并发代码的正确性产生争议（偶尔会很强烈）。他们可能会争论好几个小时，最后不可避免地终结于脆弱的细节，谁赢谁输却是无关紧要的。

必须承认的是，在那场面试中解决“双重检查锁”时，我感到很开心。最近，在听某些演讲者讲述一些快速而松散的并发技巧（通常是完全错误的）时，我想我在他们和听众脸上看到了同样的兴奋。

找到一个非常聪明的算法和一种非常快乐的感觉是很美妙的事情。如果我们能摒弃并发编程的神秘和魔法，就可以编写出更好的代码。如果并发代码的正确性是两个开发者（甚至是两个具有不同兴趣的开发者）都同意的东西，那就太好了！

读者对象

本书是为那些具有 Android 开发经验的开发者准备的。

如果你是一名新手开发人员，你可能会发现一些不熟悉的术语和概念。如果你是正在开发自己第一个 Android 应用程序的开发人员，你可能更关心如何简单地熟悉 Android 框架。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

如果你属于上述类型中的一种，本书暂时不适合你！如果真是这样，我建议你先把本书放在一边，去学习一些基础的东西，当你完成了一个完整的 Android 应用程序后再阅读本书。

我写这本书是为了让读者阅读的，虽然这听起来有点像是废话。它既不是一本教科书，也不是一本参考手册。我鼓励你自己试一下示例代码。这些示例只是沙箱实验，你可以试着扩展它们。你可以借此进行新实验，以形成自己对 Android OS 细节的理解。然而，我不希望你把这本书放在笔记本旁边来从中获取价值。

我有一段时间着迷于 Perl。回想一下，我认为主要的原因是发现那本“骆驼书”《Learning GrassMudHorse Programming Language》太有意思了（不是粉色的那本，是蓝色的那本——第 2 版）。我回想起很多远离键盘的快乐时光，脑子里没有具体的应用程序，只是单纯地阅读那本书。

我不是要拿自己和 Larry Wall 比较，也不是要用什么方式比较 Android 和 Perl。我只是希望你能喜欢阅读这本书。我希望这本书会是你坐飞机或者长途通勤路上的好伙伴。

本书内容结构

本书的前 3 章为你提供了一次复习机会。

我建议你至少不要跳过第 1 章。我在这一章介绍了一个非典型的并发模型，这个并发模型是后文讨论的基础。

我故意将第 2 章和第 3 章设计得非常短。这两章是对一些基本概念的复习，并给你提供了一个重新认识一些常见用法的机会。有经验的开发者可以选择跳过这两章。

第 4 章是一个警示故事。

本书的核心是第 5 章～第 7 章。这 3 章深入探讨了 Android 操作系统的细节。

第 8 章是“餐后甜点”—— 一些并发工具的指南。

示例代码

本书示例中的大部分代码可以在 GitHub 上找到。如果你在实验这些代码的过程中发现了一些有趣的东西，一定要和其他人分享。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负

5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF

6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅

7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

为方便下载、更新和修正，我建议你在 informit 官网上注册自己所购买的这本书。要开始注册过程，请先登录或者创建一个账户。输入 ISBN 9780134177434 并提交。一旦流程完成，你就可以在“已注册产品”中找到可用的奖励内容。

本书体例

本书中使用的印刷规范如下。

- 黑体字表示新名词、URL、E-mail 地址、文件名和文件扩展名。
- 等宽字体用于程序清单和段落中引用的程序元素，例如变量或函数名、数据结构、数据类型、环境变量、语句和关键字。
- 等宽黑体字表示注释或者应该由用户输入的其他文本。
- 加框的内容表示一个小窍门、建议或者一般注意事项。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

如果您发现错误，请登录异步社区，搜索到本书页面，单击“提交勘误”，输入相关信息，单击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，赠送给您异步社区的 100 积分（积分可用于在异步社区兑换优惠券，或者用于兑换样书或奖品）。

扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们，有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/ selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权利的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区” 是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书

和相关学习产品，为作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

致　谢

本书能够出版要感谢很多人。我的老同事Zigurd Mednieks提议我写这样一本书。Pearson Technology Group出版社的主编Laura Lewin给它开了绿灯，Jelen Publishing的文稿代理人Carol Jelen促成了这本书的出版。

Pearson的制作人员灵巧地将手稿变成了一本书。插图画家Jenny Huckleberry修整了本书的图形，开发编辑、文字编辑Abigail Manheim Bass和执行编辑Lori Lyons修改了手稿。编辑助理Olivia Basegio和项目经理Ellora Sengupta和我们一起完成了这个过程。在此致以我诚挚的谢意。

非常感谢主编 Laura Lewin，谢谢她耐心地鼓励我坚持下去。

虽然我是本书的唯一作者，但我一度想把技术编辑 Joe Bowbeer、Thomas Kountis 和 Zigurd Mednieks 列为合著者，因为他们给了我太多的帮助。他们就是技术审查团队的完美成员。作为这个领域公认的专家，他们花时间仔细阅读、理解，然后建议我进行大大小小的修改。如果本书还有错误，这些错误都是我造成的，而表达的清楚和准确都要归功于他们。我非常有幸能得到他们的帮助。

最后要感谢我的妻子 Catherine，感谢你容忍我又在沙发上戴着耳机度过了一个周末。我非常爱你，宝贝！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负

5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF

6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅

7、请于下载PDF后24小时内研究使用并删掉本PDF

目 录

第 1 章 了解并发 …… 1

1.1 并发很难 …… 1

1.1.1 软件中的并发 …… 1

1.1.2 硬件中的并发 …… 3

1.2 并发很简单 …… 4

1.2.1 线程 …… 4

1.2.2 原子执行 …… 5

1.2.3 可视化 …… 6

1.3 小结 …… 7

第 2 章 Java 并发 …… 8

2.1 Java 线程 …… 8

2.1.1 Thread 类 …… 9

2.1.2 Runnable …… 10

2.2 同步 …… 11

2.2.1 互斥 …… 11

2.2.2 常见的同步错误 …… 15

2.3 volatile …… 17

2.4 wait 和 notify …… 21

2.4.1 wait …… 21

2.4.2 notify …… 22

2.5 并发包 …… 23
2.5.1 安全发布 …… 24
2.5.2 executor …… 26
2.5.3 future …… 27
2.6 小结 …… 28
第 3 章 Android 应用程序模型 …… 29
3.1 生命周期和组件 …… 29
3.1.1 进程优先级 …… 31
3.1.2 组件生命周期 …… 33
3.2 Android 应用程序作为 Web 应用程序 …… 34
3.3 Android 进程 …… 35
3.3.1 应用程序启动 …… 35
3.3.2 Android 主线程 …… 38
3.4 小结 …… 39
第 4 章 AsyncTask 和 Loader …… 40
4.1 AsyncTask 体系 …… 40
4.1.1 AsyncTask 基础 …… 44
4.1.2 AsyncTask 的执行 …… 47
4.1.3 AsyncTask 的完成 …… 49
4.2 使用 AsyncTask …… 52
4.2.1 AsyncTask：被认为是危险的 …… 52
4.2.2 使之正确 …… 59
4.3 Loader 和 CursorLoader …… 60
4.4 AsyncTask：哪里出错了？ …… 68
4.5 小结 …… 69

第 5 章 Looper/Handler ····· 71
5.1 Looper/Handler 简介 ····· 71
5.1.1 Looper/Handler 的基础 ····· 73
5.1.2 委托执行 ····· 74
5.2 一些细节 ····· 83
5.2.1 Handler 和消息 ····· 83
5.2.2 启动 Looper ····· 86
5.2.3 Native Looper ····· 88
5.2.4 调度和同步屏障 ····· 89
5.3 小结 ····· 91
第 6 章 Service、进程和 Binder IPC ····· 93
6.1 Service 的基础知识 ····· 93
6.1.1 启动型 Service 的要点 ····· 95
6.1.2 绑定型 Service 的要点 ····· 97
6.2 Intent ····· 98
6.3 IntentService ····· 101
6.4 绑定型 Service ····· 105
6.4.1 一个简单的绑定型 Service ····· 106
6.4.2 绑定一个 Service ····· 107
6.4.3 解除对 Service 的绑定 ····· 109
6.4.4 绑定多个 Service ····· 110
6.4.5 Service 的生命周期 ····· 112
6.4.6 优先级和标志 ····· 114
6.4.7 本地绑定型 Service ····· 115
6.5 进程间通信 ····· 118

6.5.1 Parcelable ····· 119
6.5.2 Messenger ····· 120
6.5.3 使用 AIDL ····· 123
6.5.4 创建进程 ····· 128
6.6 浅谈 Binder ····· 130
6.6.1 Binder 线程 ····· 130
6.6.2 Binder 数据传输限制 ····· 130
6.6.3 绑定到死 ····· 131
6.7 小结 ····· 131

第 7 章 定时任务 ····· 132

7.1 任务特性 ····· 132
7.1.1 线程安全 ····· 133
7.1.2 生命周期感知 ····· 133
7.1.3 巧妙使用进程优先级 ····· 133
7.1.4 电量节约 ····· 134
7.1.5 记分卡 ····· 135
7.2 Timer 和 TimerTask ····· 135
7.3 Looper/Handler ····· 135
7.4 基于自定义服务的调度程序 ····· 138
7.5 Alarm Manager 和 Intent Service ····· 139
7.5.1 AlarmManagerService ····· 141
7.5.2 可调度的任务 ····· 145
7.6 同步适配器 ····· 148
7.6.1 定义同步适配器 ····· 149
7.6.2 同步适配器的工作原理 ····· 152

7.6.3 实现同步适配器 …… 154
7.6.4 对同步适配器评分 …… 159
7.7 JobScheduler …… 162
7.7.1 调度任务 …… 163
7.7.2 运行任务 …… 166
7.7.3 任务执行 …… 168
7.7.4 对 JobScheduler 评分 …… 172
7.8 小结 …… 173

第 8 章 并发工具 …… 174

8.1 静态分析 …… 174
8.1.1 Android Studio …… 175
8.1.2 Findbugs …… 176
8.2 注解 …… 183
8.2.1 JCIR 注解 …… 184
8.2.2 支持库注解 …… 184
8.3 断言 …… 185
8.4 结束语和最佳做法 …… 186

参考文献 …… 188

第 1 章
了解并发

我们建议用延迟 τ 作为可信赖的同步设备各部分功能的绝对时间单位。

约翰·冯·诺依曼

要构建正确的并发 Android 程序，开发人员需要有一个好的并发进程模型，那么这些模型是如何工作的？它们的用途是什么？对于大多数人来说，并发并不是什么大问题。对于任何多细胞动物（甚至是对病毒）来说，并发只是正常的存在。也只有我们这些沉迷于计算机的人，才会再三考虑一边走路一边嚼口香糖的想法。

1.1 并发很难

一边走路一边嚼口香糖在约翰·冯·诺依曼博士的世界里并非易事。1945 年，他在论文《The First Draft Report on the EDVAC》（冯·诺依曼，1945）里描述了最早的电子数字计算机的体系结构。70 多年来，这个体系结构几乎没有多大变化。大致来讲，纵观它们的历史，数字计算机这个巨大的状态球随着时间的推移被一系列精确定义的操作所转换。时间和顺序是机器定义的内在组成部分。

大多数计算机科学一直在讨论将一种机器状态转换成另一种更理想状态的巧妙操作序列。由于现代机器通常有超过 10^{14} 种可能的状态，因此很难对所有状态进行管理。如果转换发生的顺序可以改变，讨论必然会扩大到包括所有可能状态的所有可能组合，这将是完全不可能的。顺序执行是王道。

1.1.1 软件中的并发

当然，计算机语言是为人类编写的。它们的目的是帮助人们高效、正确，甚至以

未来读者能够理解的方式来表达一种算法（转换机器状态的指令序列）。

早期的编程语言本质上是硬件的扩展。甚至在今天，很多编程语言最初设计只是用于控制机器体系结构的映像。它们绝大多数是**过程化**的，由用于修改（**突变**）内存状态的指令列表组成。由于很难对内存的所有状态进行推理，因此随着时间的推移，语言对允许开发人员表达的状态变化日渐严格。查看编程语言设计历史的一种方法是找出一种允许开发人员轻松表达正确算法的系统，而不是表达不正确的算法。

最早的语言是机器语言，即将计算机指令一对一地翻译的代码。机器语言不可取的原因有两个：首先，甚至表达一个非常简单的想法可能都需要几十行代码；其次，很容易表达错误。

随着时间的推移，为了限制和管理程序修改状态的方式，语言缩小了选择范围。例如，大多数语言都从限制程序执行指令之间的任意跳转变为使用现在熟悉的条件、循环和过程调用。模块和 OOP（面向对象的编程）遵循的方式是将程序分解成小的、容易理解的块，然后限制这些块的交互方式。这种模块化、积木式的方式让现代语言更抽象且更富有表现力。有的语言甚至有完善的类型系统——有助于防止错误。然而，修改机器状态的指令列表仍然是必需的。

1. 函数式编程

虽然大多数计算机研究者和开发者关注的都是在基于冯·诺依曼体系结构的硬件上做更复杂的事情（这些硬件体积更大、速度更快），但一个小而执着的团队在追求一种完全不同的想法——**函数式编程**。

纯函数式编程和过程化编程的区别在于它没有可变状态。函数语言不是对机器状态的连续变化进行推理的，而是求解给定参数的函数值。这是一种相当激进的想法，需要思考才能理解它是如何工作的。然而，如果可以从并发角度看，它有一些非常吸引人的方面，尤其是如果没有可变状态，就不存在隐式时间或顺序。如果没有隐式顺序，并发就只是一个无趣的同义反复。

在创建第一个广为大众所接受的过程语言 Fortran 仅一两年后，John McCarthy 于 1958 年引入了第一个函数语言 Lisp。从那时起，Lisp 等函数语言（Scheme、ML、Haskel、Erlang 等）就被视为卓越但不切实际的教育工具或者时髦开发人员的口头禅。由于摩尔定律（摩尔，1965）更容易预测一个芯片上的处理器数（而不是单个处理器的速度），因此人们不再对函数语言不屑一顾［到 1975 年，摩尔修正了最初的想法，让这一概念正式化，他说集成电路（IC）的数量每两年翻一番］。

函数式编程是并发编程的一个重要策略，可能在将来变得更重要。Android 使用的语言 Java 不适合作为函数式语言，当然不支持与大多数语言相关的复杂类型系统。

2. 语言作为约定

不管是函数式还是过程化，编程语言都是抽象的。现在只有一小部分开发人员需要接近机器语言。即使是这一小部分开发人员，他们也可能是在由软件虚拟机或者芯片固件实现的虚拟指令集上编写代码。唯一可能想要详细了解特定硬件指令集精确行为的开发人员是为其编写编译器的开发人员。

因此，用某种语言编写程序的开发人员期望通过对该语言断言来了解该程序的行为。开发人员会对编写程序的语言（抽象）进行推断，并且几乎不需要通过检查实际的机器代码来证明程序是正确的（或者不正确的）。例如，他可能会推断某件事情发生 14 次的原因是循环计数器初始化为 13，在每次循环时递减 1，并且循环在计数器为 0 时终止。

这非常重要，因为大多数语言都是命令式（而不是函数式）抽象。即使是硬件、寄存器、高速缓存、指令管道和时钟周期，通常也都不会在程序设计中出现，但当我们对程序进行推理时，仍然会对序列进行推理。

1.1.2 硬件中的并发

讽刺的是，过程化语言最初是控制体系结构的反映，而不代表计算机硬件的行为。虽然早期计算机的 CPU 可能只在其内部时钟的每个节拍进行一次操作，但所有现代处理器都同时执行多个任务。在等待单个操作完成的过程中，闲置一个 40 亿门集成电路上的 1/4 晶体管毫无意义。

为什么一切都有顺序？

顺序执行可能是人类理解事物的固有部分。也许我们先将它强加给硬件设计，然后在语言设计中延续，因为它是我们思维方式的反映。

硬件是物质的东西。它是现实世界的一部分，而现实世界绝对不是连续的。现代的硬件是并行的。

除了在并行处理器上运行，现代程序越来越频繁地与一个疯狂并行的世界交互。即使是具有普通功能的手机用户也在不断地进行多个任务：一边浏览网页一边听音乐，

或者接听突然打来的电话。同时，传感器、硬件按钮、触摸屏和传声器都同时向程序发送数据。维持“顺序性”的错觉很有挑战性。

开发人员处于奇怪的位置。如图 1.1 所示，他正在与并行世界交互的程序中，为在一个高度并行的处理器上运行的顺序抽象构建一个指令序列。

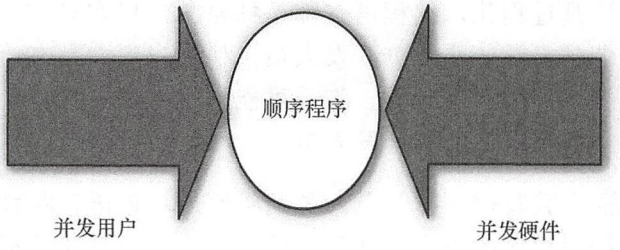


图 1.1　并发世界中的顺序程序

1.2　并发很简单

到目前为止，讨论的是重新构建并发的思路，并发不是让程序运行得更快的方式，也不是程序员执行多个任务的复杂把戏。相反，程序的明显顺序执行才是复杂的技巧。顺序执行是编译器编写者和硬件架构师维护的假象。并发是对所制造约束的放松。

1.2.1　线程

在开发人员的环境中，时间和顺序是严格且隐含的约束，“并发”只是“未指明顺序”的另一种说法——到处都有事情发生的方式。一个并发程序只是向已并发的世界宣布其正确性不依赖于事件在单独组件中发生的顺序。在并发程序中，那些独立、部分有序的组件称为**执行线**程或者**线程**。在一个线程中，指令仍然按照严格的顺序执行。但在两个单独的线程中，指令执行的顺序完全未指定。

在早期的计算中，线程作为并发模型的选择并不明显。需要无序执行的开发人员被迫自行生成并发结构。20 世纪 60 年代的文献和代码都包含各种各样的异步执行模型。

线程可能起源于 20 世纪 60 年代末 IBM 的 OS/360。它们称为“任务”，是一种操作系统级服务，为开发人员省去了自己构建并发抽象的麻烦。1991 年，Java（当时称为 Oak）采用线程模型并在语言中支持它，这时操作系统还不支持线程。

即使在今天，线程也不是唯一的并发模型。例如，Erlang、Go 和 Clojure 等语言都使用完全不同的模型。

在编程模型中引入线程并不存在实质问题。并行操作两辆车没有任何问题，除非它们尝试同时占用同一个空间。同样，操作两个完全独立的线程也完全是安全的。此时此刻，有几百万个程序同时在几百万台独立计算机中运行，其中每个程序都在自己的执行线程中并发运行。其中的大多数程序不以任何方式进行交互，并且它们的行为定义得很好。只有当线程需要共享状态和资源时才会出现问题。

1.2.2 原子执行

当多个线程修改同时访问的状态时，结果就很容易变得不确定。由于语句的执行顺序之间没有关系，因此时序的细微变化会改变程序运行的结果。

考虑下面的代码：

```
executionCount++;
someTask();
```

只通过查看，会认为 executionCount 是计算函数 someTask 被调用的次数。但在并行环境中，这段代码其实没有确定的行为，因为++操作不是原子的——它不是单一、不可分割的行为。表 1.1 演示了一个无法记录一次执行的执行序列。

表 1.1 非原子执行

executionCount = 4	
线 程 1	线 程 2
读取执行计数（4）	读取执行计数（4）
递增（5）	递增（5）
存储执行计数（5）	存储执行计数（5）
调用 someTask	调用 someTask

同步是多个 Java 线程共享状态的基本机制，这样的交互结果就是确定的。同步本身是一个非常简单的想法：一段由互斥锁或**互斥量**保护的代码**临界区**。当一个线程进入临界区（也就是说，它开始从此处执行指令）时，就称为抢占互斥量。在第一个线程离开前，没有线程可以进入临界区。

只有当任何时候都只允许一个线程访问临界区时，前面的代码才是确定的：

```
synchronized(this) {
    executionCount++;
    someTask();
}
```

同步是创建正确的并发 Java 程序的关键，并且是很多非简单事情的基础。本书后续章节将介绍这些内容。

1.2.3　可视化

然而，还有一件很简单的事情需要记住，上面的代码是抽象的！它是用计算机语言（这里是 Java）编写的，因此与编译器编写者、JVM 开发人员和硬件架构师编写的硬件实际行为有关。这两种 Java 语句翻译成上百条微指令，其中很多微指令在几十个硬件时钟周期中并行执行。这两条语句按顺序执行只不过是一种假象。

维持这种假象并不是底层开发人员自然而然做的事情。相反，他们发现顺序程序幼稚、笨拙且浪费时间。他们非常乐意通过重新排列指令、并行执行多个指令和将一种程序状态表现为多个副本等方式来修复这些程序。这样，他们就能尽全力充分利用多处理器（包括人们口袋里的微小设备）的巨大力量。

一般来说，我们很乐意让他们执行这些优化。他们通过使用应用程序开发人员不感兴趣的技巧，让程序在多种硬件平台上运行得更快。然而，这种优化有一个重要条件：它们不能破坏顺序性的假象！也就是说，编译器和硬件管道可以重新排列和并行处理所有想优化的代码，只要开发人员不说出他们这样操作即可。

让程序并发时，开发人员要清楚地指出由不同线程控制的状态之间没有顺序依赖关系。如果没有顺序依赖关系，编译器就可以任意执行所有本来不安全的优化。由于不同线程的事件之间没有明确的顺序，因此编译器可以随意地改变一个线程的执行顺序，而不用考虑其他线程中的语句。

一个正确的并发程序会遵守维持假象的约定。应用程序开发人员和硬件开发人员之间的谈判产生一种语言，而这种语言就是一个约定。应用程序开发人员得到了顺序执行的假象，这是他们可以推理的东西。硬件开发人员得到了一个工具箱，他们可以用它来让程序快速运行。处于中间的就是约定。

在 Java 中，这个约定称为**内存模型**。约定的一方是应用程序员，他会用高级语言推理程序。约定的另一方是编译器编写者、虚拟机开发人员和硬件架构师，他们会移

动所有未明确禁止的内容。在讨论并发性时，谈论硬件的开发人员没有抓住要点。一个正确的并发程序不是硬件，而是遵守约定的开发人员。

幸运的是，约定在 Java 中很容易陈述。下面的话几乎完全说明了这一点：

> 当多个线程访问一个给定的状态变量，并且其中的一个想要写入时，它们必须使用同步协调对其的访问。
>
> （Göetz 等，2006 年）

一个正确的并发 Java 程序是一个能够正好（不多不少）遵守这个约定的程序。要特别注意的是，线程是否读取或写入可变状态并不影响其同步的必要性。

1.3 小结

并发本身并不可怕，我们在现实世界中整天都在处理它，难的是为基于可变状态的计算机编写并发程序。这很难，因为顺序概念是我们推断程序的一个重要隐式基础。

并发缓和了命令式计算机语言中固有的严格顺序。Java 中的机制就是线程，一个线程执行指令的顺序与其他线程中指令执行的顺序无关。开发人员使用互斥锁（互斥）控制线程访问代码的临界区，从而限制了两个不同线程在临界区执行指令的数量。

现代大多数计算机语言都会产生顺序执行的假象。然而，它们在幕后会使劲重新排列、并行化和缓存以充分利用硬件。阻止这些优化让程序行为变得不确定的只有约定，一个正确的程序会遵守这个约定。

从来没有人说过并发容易。但它也确实很简单——只要遵守约定即可。

第 2 章
Java 并发

当多个线程访问一个给定的状态变量，并且其中的一个想要写入时，它们必须使用同步协调对其的访问。

Brian Göetz

Java 是最早包含并发的语言之一。在低级中嵌入语言是创建并发程序所必需的结构。此外，Java 可以很方便地移植到多个硬件架构上，它还第一个定义了内存模型并指定其并发构建的确定性行为。

由于 Android 的并发架构是基于 Java 的，因此本章是对 Java 基本并发机制的回顾。当然，可能需要一本书才能完整描述 Java 中的并发，所以我们的目的只是让你回想起这个概念。

注意

Java 并发编程的相关资源可参考《Java 并发编程实践》（Göetz 和 Peierls 等）。任何想要使用 Java 并发编程的人（包括所有 Android 开发人员）都应该阅读这本书。

2.1 Java 线程

在 Java 中，`Thread` 类实现线程。每个正在运行的应用程序都有一个隐式线程，通常称为“默认”或者“主”线程。产生一个新线程包含创建一个 `Thread` 类的实例，然后调用其 `start` 方法。

注意

在 Android 中，“主”线程通常也称为“UI”线程，因为它支持大多数 UI 组件。这个名字可能令人感到困惑，实际上有多个运行 UI 的线程，并且主线程甚至支持没有 UI 组件的应用程序。

2.1.1 Thread 类

start 和 run 这两个 Thread 方法非常特别。start 方法没有参数和返回值，从调用代码的角度来看，它根本什么都没有做，只是返回并将控制权传送给程序中的下一条语句。

然而，神奇的是这个调用实际上返回两次，一次是返回到程序的下一条语句，另一次返回 Thread 对象 run 方法调用产生的一个执行线程中。这个新线程在 run 方法中依次执行语句，直到该方法返回为止。当 run 方法返回时，这个新线程就终止了。

清单 2.1 是一个使用 Java 线程的简单示例。

清单 2.1 产生一个线程

```
public class ThreadedExample {
    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());
        new Thread() {
            @Override
            public void run() {
                System.out.println("running: " + Thread.currentThread());
            }
        }.start();
        System.out.println("finishing: " + Thread.currentThread());
    }
}
```

该程序在运行时会打印 3 条消息，然后终止。它定义了一个 Thread 的匿名子类，这个子类重写了其 run 方法。这个方法在一个新线程上执行，其顺序与调用 start 方法的线程执行顺序不同。

运行程序的输出清单说明了这一点：

```
starting: Thread[main,5,main]
finishing: Thread[main,5,main]
running: Thread[Thread-0,5,main]
```

要注意的是，run方法打印的消息验证了在新线程run方法中当前线程不是执行其他两条打印语句的线程。

这3条消息将以什么顺序出现？“starting”消息总是先出现。Java保证在程序顺序中新线程的start方法调用之前的任何语句和新线程的任何语句之间，都存在一个“先于发生”规则。也就是说，在新线程启动之前一切都是按照程序顺序发生的，并且这个新线程会认为它启动时才有“这个世界”的。

其余两条消息出现的顺序无法保证。它们都会打印，但Java没有定义谁先出现。对于在一些特定硬件上运行的Java实现来说，这两条消息大多数时候以某种顺序出现。然而，以这两条消息碰巧以某种顺序出现作为它们总是以这种顺序出现的证据是完全不正确的。这两条消息之间是无序的。

此外，如果Print方法不是线程安全的，那么这两条语句甚至可以是互相穿插的：运行程序产生的输出将是这两条语句的不可读组合。

2.1.2　Runnable

虽然上面的“Thread”子类清单是创建新执行线程最简单的方式，但它未必是最佳方式。由于Java对单实现继承的约束，因此在新线程上运行的任何语句都必须是Thread的子类，这是一种极大的遗憾。事实上，这不是一个要求。

当Runnable接口实现传送给一个新的线程对象时，Thread默认的运行方法就会调用Runnable，srun方法。清单2.2显示了一种与上一个清单有相同行为的不同实现。虽然无法立即在清单2.2中显现，但这个清单要灵活得多，因为Runnable可以继承任何一个超类行为。

清单2.2　带有Runnable的线程

```
public class RunnableExample {
    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());
        new Thread(new Runnable() {
            @Override
```

```
            public void run() {
                System.out.println("running: " + Thread.currentThread());
            }
        }).start();
        System.out.println("finishing: " + Thread.currentThread());
    }
}
```

2.2 同步

如上一章所述，同步是将一组语句原子化的机制：在任何给定的时间里，只有一个线程可以在同步锁控制的块（临界区）中执行语句。Java 中的同步是使用 synchronized 关键字和一个对象作为互斥锁完成的。Java 保证最多只有一个线程可以在任何时间执行给定同步块中的语句。

注意

互斥（mutex）、监视器（monitor）和锁（lock）这 3 个词在这里几乎可以互换。**互斥**可能是最清楚的定义。它是一种提供互斥的锁：每次只有一个线程可以持有它。Java 规范用**监视器**这个词描述一个实例，其相关的互斥锁用于控制一个同步块的访问。**锁**是一个更宽松的术语，具有更直观的含义。

2.2.1 互斥

虽然线程是特殊的对象，但在 Java 中任何对象都可以作为同步块的互斥来使用。每个 Java 对象都有一个监视器。

清单 2.3 演示了 synchronized 关键字的简单使用。

清单 2.3 在一个对象上同步

```
public class SynchronizedExample {
    static final Object lock = new Object();

    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());
```

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
            new Thread(new Runnable() {
                @Override
                public void run() {
                    synchronized (lock) {
                        System.out.println("running: " + Thread.currentThread());
                    }
                }
            }).start();

            synchronized (lock) {
                System.out.println("finishing: " + Thread.currentThread());
            }
        }
    }
```

清单 2.3 中的临界区是两个同步块中的两条打印语句。这个用于监视器、锁的对象没有什么特别的。它只是在代码早期创建的 `Object` 对象中一个最简单的实例。

我们可以分析一下，看看使用同步是如何影响这个清单的。毕竟，只有一个线程有机会执行匿名 `Runnable` 的 `run` 方法：在匿名线程 `start` 方法中创建的线程。同样，只有一个线程可以打印“finishing”消息：主线程。如果只有一个线程可以进入临界区，那么同步可能是无意义的。

在这个程序的非同步版本中，新线程可能会在主线程打印“finishing”消息的同时打印“running”消息。如果打印方法在内部是不同步的，那么就有可能造成这两条消息不可读。然而，由于清单 2.3 代码中的两个同步块都使用同一个互斥对象，因此这个版本的程序不会使这两条消息混合在一起。即使有两个独立的同步块，任何时候都只有一个线程可以持有一个给定对象的互斥。

其中，线程 T1 会夺取监视器对象的互斥，进入一个临界区并打印一条消息。由于它正持有互斥，因此另一个线程 T2 无法进入它的临界区。T2 会在其临界区的开始处等待锁，直到 T1 将锁释放为止。当 T1 释放锁时，就会通知 T2 可以继续了。然后，它会进入其临界区，打印消息并继续。

要注意的是，应用程序的这个同步版本仍然不会定义这两条消息出现的顺序。无法确定这两个程序线程是主线程还是新线程会先进入其临界区。像前面的清单一样，这个清单认为包含一个竞争条件。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

1. this 上的同步

由于每个对象都有一个监视器，因此在这个监视器上可以方便在一个方法中同步的对象就是 this。通过同步自身来同步访问其状态的对象，是很有道理的。

清单 2.4 是这项技术的一个示例。Runnable 仍然是一个匿名类的实例。然而在这个版本的程序中，对实例的引用保存在变量 job 中。this 上的同步（源自匿名可运行的 run 方法）和 job 上的同步在主方法中都是在同一个对象上进行的。这个版本的行为与清单 2.3 中的相同。

清单 2.4 this 上的同步

```
public class SynchronizedThisExample {
    private static final Runnable job = new Runnable() {
        @Override
        public void run() {
            synchronized (this) {
                System.out.println("running: " + Thread.currentThread());
            }
        }
    };

    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());
        new Thread(job).start();
        synchronized (job) {
            System.out.println("finishing: " + Thread.currentThread());
        }
    }
}
```

注意

你可能会注意到，清单 2.4 并不是对清单 2.3 的改进。正如清单 2.5 中所示的，同步整个方法是倒退，因为它破坏了大多数多线程的目的。

2. 同步方法

Java 有在 this 上同步的简写。在方法签名中使用 synchronized 关键字和将这

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

个方法的内容封装到在 this 上的同步块中完全相同。例如，清单 2.5 中的方法 sync1 和 sync2 是相同的。

清单 2.5　同步方法

```
public class SynchronizedMethodExample {
    private int executionCount;

    public synchronized void sync1() {
        executionCount++;
        someTask();
    }

    public void sync2() {
        synchronized (this) {
            executionCount++;
            someTask();
        }
    }

    private void someTask() {
        // ...
    }
}
```

用 synchronized 关键字限定的静态方法会同步到其类对象上。例如，清单 2.6 中的两个方法也是相同的。

清单 2.6　同步静态方法

```
public class SynchronizedStaticExample {
    private static int executionCount;

    public static synchronized void sync1() {
        executionCount++;
        someTask();
    }

    public static void sync2() {
```

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
        synchronized (SynchronizedStaticExample.class) {
            executionCount++;
            someTask();
        }
    }

    private static void someTask() {
        // ...
    }
}
```

3. 可重入监视器

对象监视器被认为是**可重入**的。虽然在任何给定的时间内只有一个线程可以持有互斥，一旦线程持有它，那么该线程会再次尝试捕获它，但这基本上都是空操作。清单 2.7 所示的是在同一个监视器 `this` 上同步两次的程序。程序可以成功运行。它是正确的，但没什么实际用途。

清单 2.7 同步静态方法

```
public class ReentrantLockingExample {

    public static void main(String... args) {
        new ReentrantLockingExample().run1();
    }

    private synchronized void run1() { run2(); }

    private synchronized void run2() {
        System.out.println("reentrant locks!");
    }
}
```

2.2.2 常见的同步错误

不正确的同步是并发程序中最常见的错误。然而，这种错误要比将正确的内容放入同步块中多得多。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

1. 使用单个互斥

只有当临界区在单个互斥上同步时，对临界区的访问才是原子的。保护代码的是互斥，而不是块。

虽然在 this 上锁定非常方便，但有经验的开发人员可能会意识到，在单个对象 this 上同步可能会序列化并发操作。例如，如果请求不相互阻塞，具有状态组件的对象可能会得到显著优化，而这些状态组件会请求更新到独立的网络端点。为了使各个操作互不干扰，这个对象可能会使用不同的锁控制独立组件的访问。然而，这可能会产生类似于清单 2.8 所示的代码。请试着找出问题所在。

清单 2.8　错误的互斥锁

```
public class LeakingLockExample {
    private String title = "";

    public void setTitle(String title) {
        if (null == title) {
            throw new NullPointerException("title may not be null");
        }
        synchronized (this.title) { this.title = title; }
    }
}
```

显然，问题是 this.title 的赋值没有被正确同步。

在这个清单中，两个不同的线程可以看到 this.title 的赋值由不同的互斥保护。线程 T1 捕获字符串上由 this.title 目前引用的互斥。然而，当它给 this.title 赋值时，线程 T2 可以进入 setTitle 方法并捕获一个不同的互斥。由于这两个线程还没有在一个互斥上同步，因此它们之间没有“先于发生”规则，并且代码能够以多种方式表现。

2. 死锁

并发程序中最常见的第二种错误是死锁。上面讨论的可重入监视器指出，尝试两次捕获同一个互斥不会挂起的线程。然而，有两个或者多个互斥很可能会创建出即使在正确的情况下，没有线程也可以继续前进的代码。

如果两个或者多个线程都持有另一个线程所需要的锁，就会发生死锁。例如，如果线程 T1 和 T2 都需要访问分别由锁 L1 和 L2 保护的资源 R1 和 R2，就可能发生这种

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

情况。如果在 T2 捕获并 L2 访问 R2 的同时 T1 捕获 L1 并访问 R1，T1 和 T2 就不会继续前进。T1 需要 L2，但它被 T2 持有，所以无法获取。T2 需要 L1，但它被 T1 持有，所以也无法获取。清单 2.9 是一个可能会发生死锁的程序清单。

清单 2.9 死锁

```
public class DeadlockingExample {
    static final Object lock1 = new Object();
    static final Object lock2 = new Object();

    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());

        new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lock1) {
                    synchronized (lock2) {
                        System.out.println("running: " + Thread.currentThread());
                    }
                }
            }
        }).start();

        synchronized (lock2) {
            synchronized (lock1) {
                System.out.println("finishing: " + Thread.currentThread());
            }
        }
    }
}
```

2.3 volatile

大多数开发人员知道在并发程序中需要原子性，也了解如何使用同步来保证它很容易实现。正如第 1 章所讨论的，一种更常见的误解与可见性有关。这种误解经常表现在它相信必须要同步写可变状态，而不读取它。清单 2.10 就是这种错误的

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

一个示例。

清单 2.10　不正确的同步

```
public class IncorrectSynchronizationExample {
    static boolean stop;

    private static final Runnable job = new Runnable() {
        @Override
        public void run() {
            while (!stop) { // !!! Incorrect!
                System.out.println("running: " + Thread.currentThread());
            }
        }
    };

    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());

        new Thread(job).start();

        synchronized (job) { // !!! Incorrect!
            System.out.println("finishing: " + Thread.currentThread());
            stop = true;
        }
    }
}
```

这段代码是不正确的。只有一个线程使用同步，所以无法完成任何事情。

这段代码背后的推理通常是这样的："run 方法中循环执行的次数并不重要。不管它运行 2 次或者 7 次，在它注意到停止标志前，没有人会关心……如果没有同步，它将运行得更有效。"

然而，要记住的是，本章开头对 Java 约定的简单描述指出，任何线程中的写访问都可能会潜在影响同步的需要。在主线程上对 stop 赋值和在产生线程上从 stop 上读取之间，不存在任何类型的"先于发生"关系。一个原因不一定要导致另一个结果。

作为一个可能失败的程序清单，请考虑代码的编译版本，其中变量 stop 表示为一个寄存器。现在假设这两个线程在 CPU 的两个独立的内核上运行。它们可能会有

非卖品，仅供非商业用途或交流学习使用

两个完全不同的停止标志，每一个都是一个内核的寄存器。只有当生成的代码强制这两个值同步时，一个值的改变才对另一个值可见。这种同步是昂贵的，如果没有必要的指示可能会被优化掉。产生的线程能够运行到它被外部终止时为止。

再次重申，正确的并发是关于约定的。虽然清单 2.10 讨论了硬件和编译器来演示错误，但可以通过引用已知的硬件和编译器架构是错误的推理来证明不存在错误。内存模型就是法则，若不遵守它，程序就是不正确的。

清单 2.10 中的可见性问题可以通过关键字 volatile 来解决。标记为 volatile 的变量表现得像是对它的每个引用都被封装在一个同步块中。清单 2.11 显示的是已更正的程序。

清单 2.11 Volatile

```
public class VolatileExample {
    static volatile boolean stop;

    private static final Runnable job = new Runnable() {
        @Override
        public void run() {
            while (!stop) {
                System.out.println("running: " + Thread.currentThread());
            }
        }
    };

    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());

        new Thread(job).start();

        System.out.println("finishing: " + Thread.currentThread());
        stop = true;
    }
}
```

与清单 2.2 和清单 2.3 一样，如果消息打印不是同步的，这个程序可能会混淆消息。然而，如果主线程修改了停止标志的状态，产生的线程就能保证看到这个更改。

非卖品，仅供非商业用途或交流学习使用

要注意的是，让停止标志成为 volatile 的并不能保证程序会终止！然而在这个版本中，没有任何语句能保证停止标志将设置为 true。由于这两个线程之间没有指定的顺序，因此一个完全合法的场景是：第二个线程开始，而第一个线程暂停并且永远不会恢复。在这个场景中，第二个线程“**饿死了**”第一个线程。

当没有足够的处理器推动程序中的所有线程时，某种调度程序必须将这些处理器分配给线程。希望它能以某种方式给每个线程“公平份额”。实际上，自从有了 Android 平台的调度程序后，Java 宽松的调度规则很少被关注。然而即使在 Android 中，“公平份额”的确切含义有时也令人惊讶。

还要注意的是，虽然提供了可见性，但 volatile 关键字没有提供原子性。清单 2.12 所示的程序可能无法准确地打印 11 条消息。正如上一章所描述的，++操作符不是原子的，读-改-重写可能会保存一个不正确的值。

清单 2.12　不正确的 volatile

```
public class VolatileExample {
    static volatile int iterations;

    private static final Runnable job = new Runnable() {
        @Override
        public void run() {
            while (iterations++ < 10) {
                System.out.println("running: " + Thread.currentThread());
            }
        }
    };

    public static void main(String... args) {
        System.out.println("starting: " + Thread.currentThread());

        new Thread(job).start();

        while (iterations++ < 10) {
            System.out.println("finishing: " + Thread.currentThread());
        }
    }
}
```

2.4 wait 和 notify

Java 中还包含两个并发原语：wait 方法和 notifyAll 方法。notifyAll 方法还有一个不常用的变体 notify。

2.4.1 wait

wait 方法允许一个通过捕获互斥进入同步块的线程在块中暂停并释放这个互斥。这不违反原子性，因为虽然线程在同步块内部，但它是挂起的，没有执行。

只有当前线程持有调用 wait 方法的对象的互斥时，wait 方法才可以被调用。清单 2.13 会抛出一个 java.lang.IllegalMonitorStateException，因为只有执行线程不持有监视器对象 lock 时，wait 方法才会被调用。

清单 2.13 IllgalMonitorStateException

```
public class WaitExceptionExample {
    private Object lock = new Object();
    public static void main(String... args) {
        try { lock.wait(); }
        catch (InterruptedException e) {
          throw new AssertionError("Interrupts not supported");
        }
    }
}
```

当一个线程在同步块中执行，并且它在这个块的监视器对象上调用 wait 方法时，线程就会挂起并释放这个监视器。在调用 wait 时，挂起的线程可以被后续的 notify 或 notifyAll 调用并重新调度。当然，这个调用必须由另一个线程执行，挂起的线程在重新调度前不能做任何事情。

清单 2.14 所示的代码包含两个同步块，它们一起执行来控制调用 req.run 的线程数。

当一个新线程（这次我们将它称为 Edsger）进入第一个临界区时，代码会检查是否已经有了太多请求正在运行。如果没有，Edsger 就会递增运行计数，离开临界区并运行它的请求。当完成请求时，它就会尝试进入第二个临界区。

如果其他线程正在执行其中的一个临界区，Edsger 就必须等待其他线程离开或者调用 wait。一旦发生这种情况，Edsger 就会进入第二个临界区，递减运行请求的计数，并退出这个方法。

然而，考虑一下当 Edsger 进入第一个同步块时，若已经有 3 个其他线程（可能是 Tony、Niklaus 和 Per Brinch）正在运行请求，由于运行计数已经达到最大值，因此 Edsger 就会进入循环，只有当运行计数小于最大值时，Edsger 才能离开这个循环。运行计数小于最大值的唯一方法是：当前运行请求的其他线程必须进入第二个同步块中，并递减计数。由于 Edsger 会在调用时释放监视器，因此其他线程可以做到这一点。

注意

关于线程中断处理复杂主题的完整讨论不属于本章所述范畴。在本章的清单中，中断是不允许的。

2.4.2　notify

然而，另一个线程递减运行计数不会重启 Edsger。一旦线程等待了一个特殊的监视器，即使它能够运行，也不会被再次调度，除非某个运行的线程为这个监视器调用 notifyAll 或者 notify。清单 2.14 在第二个临界区中进行这样的操作。

只有当运行计数达到最大值时，才会有线程在第一个临界区中等待，因此第二个临界区中的 notifyAll 只有在那时才会被调用。

了解 notify 和 notifyAll 这两个方法之间的区别非常重要。调用 notify 会重新安排一个等待监视器的线程，notifyAll 会重新安排所有等待监视器的线程。当不同的线程由于不同原因而等待时，这一点非常重要。

清单 2.14　使用 wait 和 notify

```
public static final int RUN_MAX = 3;
private final Object lock = new Object();
private int running;

public void rateLimiter(NetworkRequest req) {
  synchronized (lock) {
    while (running >= RUN_MAX) {
      try { lock.wait(); }
      catch (InterruptedException e) {
```

```
            throw new AssertionError("Interrupts not supported");
          }
        }
        running++;
      }
      try { req.run(); }
      finally {
        synchronized (lock) {
          if (running-- == RUN_MAX) { lock.notify(); }
        }
      }
    }
```

考虑一下，如果修改清单 2.14 中的代码以启用两个不同的运行最大值，一个用于执行读请求的线程，另一个用于执行写请求的线程，其结果会怎么样？假设读者的运行计数小于最大值，就会强制调用 notify。一种完全可能的情况是：虽然读运行计数小于最大值，但写运行计数仍然大于最大值，这时只有一个等待写入的线程会重新调度。写线程可能会在未通知其他线程的情况下立即再次等待。在这种情况下，没有读线程（没有可以前进的线程）被重新调度。

在这种情况下，当不同的线程在等待不同的内容时，调用 notifyAll（而不是 notify）是必不可少的。顾名思义，前者会重新调度所有等待监视器的线程。这是比较昂贵的：唤醒每个线程，检查环境，并前进或者再次调用 wait（如果它没有获得所需资源）。正如清单 2.14 所示，只有当线程无法区分时，使用 notify 才是安全的。然而，即使是这样，最好还是使用 notifyAll 作为未来使用的一种方式。

2.5 并发包

虽然了解 Java 的底层并发结构很有用，但大多数代码不会使用这些结构。大多数代码关心的都是如何实现更高级的目标。对于大多数应用程序来说，设计 wait 和 notify 就像是在考虑汽化器设计的情况下，计划从旧金山到波士顿的旅行。

在 Java 5 中，Java 在 java.util 包中引入了一个包含更高级并发抽象的框架。和前面的集合框架一样，并发框架是安全且强大的，并且足以满足大多数编程任务。包含调用底层结构（例如 wait 和 notify）的代码可能会出错。

严禁网络传播本PDF，违者责任自负！

2.5.1　安全发布

并发编程的一个关键问题是线程之间的数据共享。将数据正确地从一个线程传输到另一个线程称为**安全发布**。仔细检查本章的规则就可以找到安全发布的几种策略。

最简单的策略是不可变的数据。如果任何线程都不可能改变数据状态，那么并发规则的前提是不适用的，共享数据是安全的。一般来说，Java 中不可变的状态应该声明为 `final`。《Java 并发编程实践》一书引入了“有效的不可变性”来讨论所有人都不同意修改一个变量值的情况。这当然管用，然而更安全的做法是使用这种语言来保证变量值不能被修改，而不是依靠这个变量值不会被修改的保证。

在线程之间传递可变状态是一件棘手的事。幸运的是，有一种标准的习惯用法。图 2.1 所示即为一种安全发布的习惯用法。

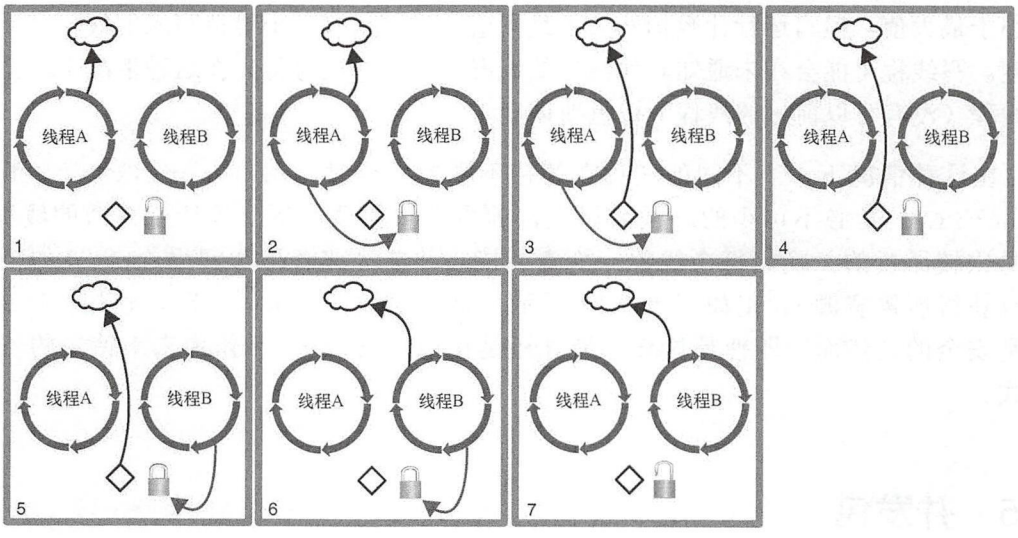


图 2.1　一种安全发布的习惯用法

当线程 A 想将一个对象传递给线程 B 时，它就会使用一个互斥锁和一个下拉框。线程 A 捕获互斥锁，然后将这个对象的引用放入下拉框中。必须确保它不会持有被转移对象的任何引用。然后线程 A 释放互斥锁。线程 B 捕获互斥锁，并从下拉框中恢复对象的引用。线程 B 必须确保在释放互斥锁前，下拉框不再持有这个对象的引用。

严禁网络传播本PDF，违者责任自负！

清单 2.15 是在 Java 中安全发布算法的部分实现。

清单 2.15 安全发布

```
private Object lock = new Object();
private T dropBox;

/**
 * Safely publish an object to another thread.
 *
 * @param obj The caller must hold <b>no other references</b> to this object.
 */
public void publish(T obj) {
  synchronized (lock) {
    if (null != dropBox) {
      throw new IllegalStateException("The drop box is full!");
    }
    dropBox = obj;
  }
}
/**
 * Receive a published object.
 *
 * @return the received object
 */
public T receive() {
  synchronized (lock) {
    T obj = dropBox;
    dropBox = null;
    return obj;
  }
}
```

在这个传输过程中，只有一种可变状态被两个线程共享：下拉框。传递对象在任何时候都无法从多个线程中进行访问。单个互斥锁控制对下拉框的访问，因此传输是安全且正确的。

这种习惯用法通常会使用队列，而不是一个简单的下拉框。线程通过将一个对象

严禁网络传播本PDF，违者责任自负！

入队来交换数据，并通过另一个线程将其出队。

2.5.2　executor

很容易理解一个应用程序如何从生成其他线程中受益。每当有可并发执行的独立任务时，生成更多执行它们的线程可以让应用程序更高效。然而，简单地生成更多线程并不意味着可以并发执行更多任务。在一个具有 4 个内核的处理器上，为计算密集型任务生成 10 个线程并不会让应用程序运行的速度加快 10 倍。

事实上，创建太多线程会让应用程序运行得更慢。线程是重量级很高的对象。在它们之间进行切换会产生开销。它们在对象分配和垃圾回收的过程中也会造成大量开销。随意简单地创建新线程并不是一个好策略。

对于在特定硬件环境中运行给定的应用程序来说，线程数有一个最佳范围。显然，这个数值与硬件支持的并发进程数有关，因此它不必完全相同。

应用程序需要一种策略来支持其所支持的线程数。通常情况下，必要的工具是另一层抽象：将任务的概念从执行它的线程中分离出来。已经从心理上实现了从物理进程（实际的并发）到线程（无序的执行）的跨越，这种区别似乎是明显的。线程是由物理 CPU 内核驱动虚拟进程的，而任务是一个工作单元。如图 2.2 所示，任务是在线程上调度的，而线程是在处理器上调度的。

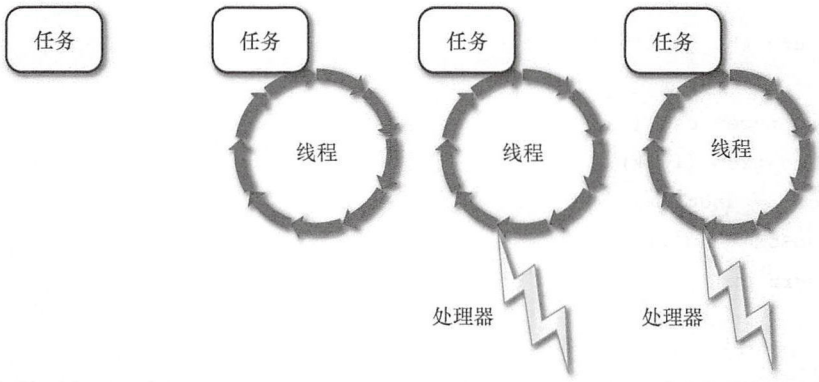


图 2.2　任务、线程和处理器

Java 5 中引入的 Java 并发框架定义了一个名为 **executor** 的服务。executor 将定义线程策略的能力和图 2.1 所示的安全发布习惯用法结合起来。一个 executor（执行器）就是为一队任务服务的一个或者多个线程池。任务关闭，进入执行器队列中换取一个 **future**，一个值的承诺。最后，任务由一个执行器线程从队列中删除并执行。执行结

严禁网络传播本PDF，违者责任自负！

果变成这个 future 的值。

executor 是从上一节的线程、同步块和通知中提取的一个很有价值的步骤。通过使用 executor，应用程序可以为最佳线程数建立一个单独、跨应用程序的策略。大多数应用程序代码只需创建新任务（即小的、相对轻量级的对象），并将它们提交给异步执行的 executor，而不关心开销和线程安全。当然，应用程序可以使用多个 executor，然而应该明确的是，有几十个 executor 的应用程序没有线程策略。

虽然 executor 是将线程策略从正在执行的任务中分离出来的抽象，但 **executor 服务**是真实世界中执行器抽象的实现。它定义了很多细节，如 executor 如何启动和停止，以及是否可以确定它处于这两种状态中的哪一个。

2.5.3 future

当一个线程 A 请求另一个异步执行某些任务的线程 B 时，B 只有几种方法可以将结果从任务中返给 A。当然，最简单的是 A 等待 B 完成，但这会错过并发点。

在 Web 应用程序中流行的另一种方法是回调。任何函数

```
f(x1, x2, x3, ..., xn) = y
```

都可以转换成一个调用

```
f'(x1, x2, x3, ..., xn, g(y))
```

也就是说，函数 f 不会返回一个值，而是将要调用的、带有返回值的函数 g 作为额外的参数。由于请求线程 A 不知道它的方法什么时候会被 g 调用，因此 g 通常会将一个任务入队，然后在某种本地队列中处理返回值。

虽然这种习惯用法很有效，尤其是返回值在网络上传输时，这可能会导致 JavaScript 开发人员经常发现的臭名昭著的“回调地域”。方法 g 可以多次在不方便的时候调用。

执行器使用的第三种策略：future，future 就是一个值的承诺。将任务交给执行器处理的客户端会接收到一个 future。这个 future 不是一个值，但客户端可以将它看作一个值，直到它真正需要 future 表示的值。如果客户端线程必须有实际值才能向前推进，那么按照定义，它就会准备好等待这个值。它调用 `Future.get` 并挂起线程，直到服务线程完成任务并将结果发布到客户端线程。这种方法大大降低了回调和异步编程的需要。

严禁网络传播本PDF，违者责任自负！

2.6　小结

本章复习了 Java 并发编程的要点，因为它们要应用到 Android 中。本章建立了几个关键点：线程对象、异步任务的 Java 表示、同步和监视器、Java 的原子性和可见性构造以及 Java 可见性的捷径 volatile。

本章还介绍了将数据从一个线程安全发布到另一个线程的常见用法，并说明了它在 Java 并发框架执行类型中的用法。executor 通过处理安全发布的多个方面并让应用程序建立一个并行性策略来降低并发编程的复杂性，这些线程足以在不产生过多调度或者内存管理开销的情况下提供最佳的并行执行。

严禁网络传播本PDF，违者责任自负！

第 3 章
Android 应用程序模型

在 Android 环境中，必须要处理生命周期超出应用程序控制的组件，这加剧了并发编程的挑战。

3.1 生命周期和组件

Android 是在移动设备无法合理支持分页内存时设计的。虽然如今大多数笔记本电脑都采用固态硬件驱动器，但 21 世纪初的闪存在大约 10 000 次写入后就磨损了。使用闪存作为通用虚拟内存系统的交换空间完全是不可能的。在仅限于物理内存的环境中，操作系统对在任何给定时间内运行的应用程序的数量有严格的上限。Android 解决这个问题的方法是允许操作系统在任何时候都可以终止应用程序进程，以便恢复另一个应用程序的内存。

注意

会突然终止应用程序的 Android 体系结构并不是唯一的可能。相反，苹果 iOS 的早期版本选择简单地限制给定时间内可以运行的应用程序数。

在一个应用程序几乎无法控制自己生命周期的环境中，应用程序作为一个可识别实体的想法就变得没有那么有意义了。相反，在 Android 世界中有组件：`Activity`、`Service`、`Provider` 和 `Receiver`。操作系统应保证在需要这些组件时，它们就在附近。在必要时，运行它们的副作用是产生一个新的应用程序进程。

注意

本章对 Android 应用程序模型和组件生命周期的介绍依赖于对 Android 组件及其功能的熟悉程度。这里不会介绍它们。有很多详细描述它们的资源，例如：Griffiths、Dawn 和 David 的《Head First Android Development》第 1 版，Mednieks、Dornin、Meike 和 Nakamura 的《Programming Android》。

严禁网络传播本PDF，违者责任自负！

虽然在很多情况下，术语“应用程序”大致等同于“进程”，但在 Android 中，应用程序只是可协调组件的集合。下面给出两个示例加以说明。

当一台 Android 系统的手机接到一个电话时，不管它在干什么，系统都会通过显示一个允许用户接电话的页面来做出响应。为了实现这一点，电话子系统会发送一个**Intent**。Intent 是描述所需服务的一个小数据包。

操作系统将 Intent 需求和支持这些需求的组件（在这种情况下，是显示一个“来电”页面的 Activity）相匹配。要注意的是，Intent 中表示的需求不是运行一个特定应用程序的请求。它是对组件的请求，这种情况是显示一个页面的组件：Activity。如果显示 Activity 必须启动一个应用程序进程，系统就会将它当成一个副作用。它的目标是显示一个页面。启动应用程序是达到这个目标的一种手段。

第二个示例与第一个示例相反。当 Android 设备启动时，该系统在启动过程中做的最后一件事情是广播 Intent（`BOOT_COMPLETED`），表明它准备开始正常运行。名为 Receiver 的组件通过注册可以接收这个特殊的广播。每个为广播注册的 Receiver 都会在系统启动不久后收到它。

有趣的是当它们接收到广播时，包含 Receiver 的某些应用程序什么都不会做。包含这种 Receiver 的应用程序完全取决于副作用。系统在开机时开机这些应用程序，因此它可以将 Intent 传递给 Receiver 组件。然而，应用程序的目标是在系统每次开机时启动。

上面的示例描述了对两种不同 Intent 的处理：一种是启动 Activity 响应一个硬件事件，另一种是启动 Receiver 响应一个系统事件。使用一个包含样例组件、Activity 和 Receiver 的不可能存在的应用程序来说明生命周期。

当系统启动时，Android 会尝试将 `BOOT_COMPLETED` Intent 传递给应用程序的 Receiver。要做到这一点，它必须创建一个进程，将应用程序加载到进程内存中，并创建 Receiver 对象的一个实例，然后调用 Receiver 的 `onReceive` 方法并传递 Intent。一旦 onReceive 方法返回，Receiver 实例就不会再使用，它会被释放，并且可以作为垃圾回收。

如果一个电话在系统开机不久后打进来，Android 就必须将这个电话 Intent 传递给应用程序的 Activity。它发现应用程序已经在运行（根据最近传递的 `BOOT_COMPLETED` Intent），因此它只需要创建所需的 Activity 实例并在其整个生命周期中运行它。

如图 3.1 所示，现在有 3 个生命周期要跟踪：Receiver、Activity 和应用程序进程。

严禁网络传播本PDF，违者责任自负！

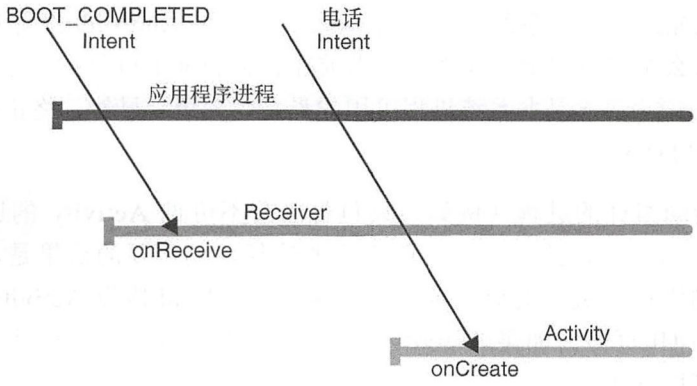


图 3.1 启动 Android 进程

3.1.1 进程优先级

应用程序进程的生命周期可能会影响这两个组件的生命周期。假设系统确定没有足够的内存执行某些所需的任务。当然这是不可能的，因为系统刚刚启动。考虑这个案例只是为了讨论。

由于需要内存，因此系统会检查运行进程来找到终止的最佳候选。它通过检查每个运行进程的 oom_adj 属性来做到这一点。oom_adj 是一个小整数，它的值是由系统动态管理的。较小的值表示较高的进程优先级（和较小的进程终止机会）：-16 是一个非常高的优先级，15 是一个非常低的优先级，如图 3.2 所示。

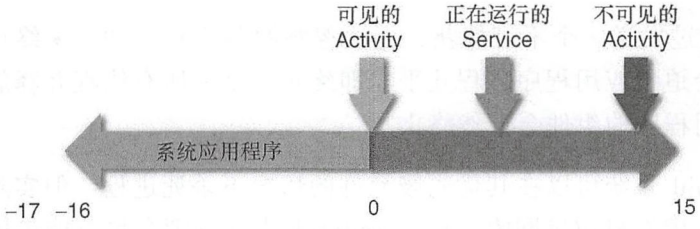


图 3.2 OOM-ADJ

注意

你可能会发现，一个进程的 oom_adj 正在带有 root 权限的设备上运行。

对习惯于使用命令行的读者可以使用 Android 调试器访问设备上的 shell。与一个给定进程相关的 oom_adj 分类为文件系统，作为文件/proc/<process-id>/oom-adj。

严禁网络传播本PDF，违者责任自负！

负数（高优先级）是为系统应用程序保留的。如果 Android 必须杀死自己的进程来恢复内存，那么系统可能就要崩溃了。当前在设备屏幕上可见的、包含 Activity 的进程有值为 0 的 oom_j，这是非系统进程可用的最高优先级。显然，终止提供当前可见屏幕的进程没有任何意义。

一个没有活动组件的进程（例如，只包含当前不可见 Activity 的进程）将会给予一个非常高的 oom_adj，给它一个可能终止的选择。其重要的后果是，一旦被隐形，应用程序被强制终止的机会就会从零变为非常高。Android 假设 Activity 组件的目的是提供一个可见的 UI 页面。如果 Activity 没有显示 UI 页面，它就不再重要了，并且支持它的进程就可以终止。

注意

Linux Kernel 的最新版本、Android 从 KitKat 开始的版本，以及使用这些内核的 API level19 都已经用 oom_score_adj 代替了 oom_adj. oom_score_adj，这样粒度要细得多，它的值介于 1000 和-1000 之间，优先级从最低到最高。虽然数字已经改变，但本质上行为是相同的。

带有运行 Service 组件的进程有一个 oom_adj，其值要比带有不活跃组件的进程小，但要比带有可见 Activity 的进程大。Service 组件永远不会像可见 Activity 那么重要，但如果它正在运行，那么它就会比不可见的 Activity 重要。例如，在应用程序的 Activity 组件都不可见的情况下，应用程序中的某个 Service 组件做有用的工作是完全有可能的。服务调度将在后面调度的应用程序策略中起重要作用。

Android 通过发送一个不可捕获、无法忽略的条信号（-9）来终止一个应用程序进程。这样做会迫使应用程序进程几乎立即终止，杀死所有线程并释放所有内存。尤其是，构成应用程序的组件会突然终止。

虽然 Android 系统可以在其优先级允许的情况下杀死进程，但实际上它非常懒。和它的“近亲”虚拟机垃圾回收一样，Android 内存管理会尽可能少地工作。如果不是立即需要内存，释放进程所做的工作就只是在浪费电量。

注意

有趣的是，虽然用 kill-9 杀死一个应用程序在大多数操作系统中只是普通的事件，但在 Android 系统中，它是终止应用程序最常见的手段。

例如，在图 3.3 所示的清单中，Receiver 和 Activity 都没有完成它们的生命周期。

Android 正在偷懒。然而，当它们的进程被杀死时，两者都会在生命周期的中期突然结束。没有警告，没有生命周期方法的调用，只是突然结束。

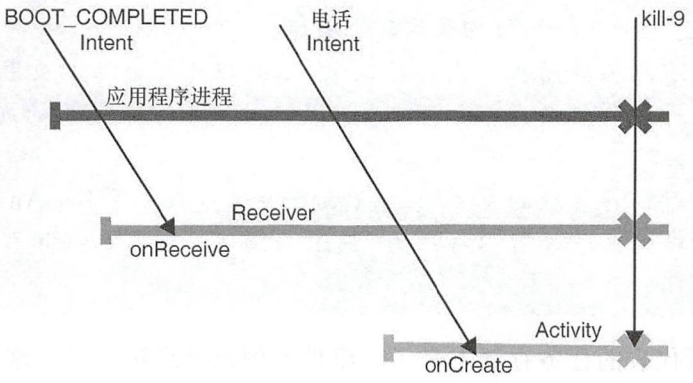


图 3.3 杀死一个 Android 进程

当然，其重要的后果是没有保存的状态消失了，没有办法找回。

3.1.2 组件生命周期

在 Android 应用程序中，单独的组件是完全独立的实体，具有完全独立的生命周期。Android 系统控制组件的生命周期，在需要时可以启动或停止它们。

Android 组件生命周期的典型示例是当设备旋转时会发生什么。Android 中非常灵活的 UI 系统支持在纵向和横向模式下完全不同的屏幕布局。当旋转设备时，很多事情都会发生改变，Android 会丢弃当前配置中为 UI 提供的全部 Activity，并为新配置创建一个新实例。图 3.4 说明了设备旋转时会发生什么。

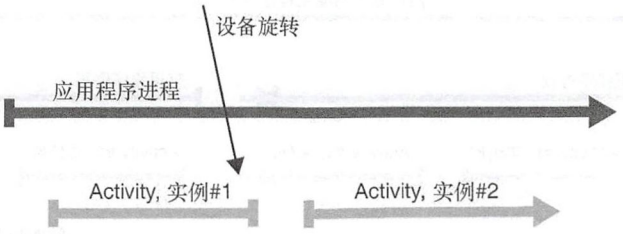


图 3.4 设备旋转

可见 Activity 运行实例的生命周期已经完成。Android 按顺序调用它的 `onPause`、`onStop` 和 `onDestroy` 方法，然后删除 Activity 的所有引用。现在它有资格进行垃

圾回收了。

> **注意**
>
> 不要混淆进程收割和垃圾回收这两个概念！
>
> 虽然两者有相似的动机（内存管理），但进程收割会结束应用程序进程，终止所有线程并释放所有内存。垃圾回收只是回收进程中未使用的内存，因此进程可以继续运行。
>
> 关联这两种内存管理的是 onTrimMemory 生命周期事件。Android 系统使用这些事件的指示应用程序有内存压力，通过收缩做出响应的应用程序不太可能被杀死并回收空间。

即使它们所代表的任务存活下来了，组件实例也会进进出出。这样很容易想象，图 3.4 中显示的 Activity 实例#2 与其前任实例#1 的行为完全相同。然而，实例#1 会消失并被遗忘，实例#2 是唯一存活的 Activity 实例。正如 Android 在需要时创建和销毁进程一样，它会在需要时创建并销毁组件实例。

再次声明，这些操作会有一些重要的后果：

- 没有保存状态的组件会丢失它；
- 代码不能依赖它刚刚看到的组件对象存在。

3.2　Android 应用程序作为 Web 应用程序

图 3.5 所示的是 Android 应用程序生命周期的另一个更完整的描述。这次有更多组件和用户感知了应用程序生命周期。

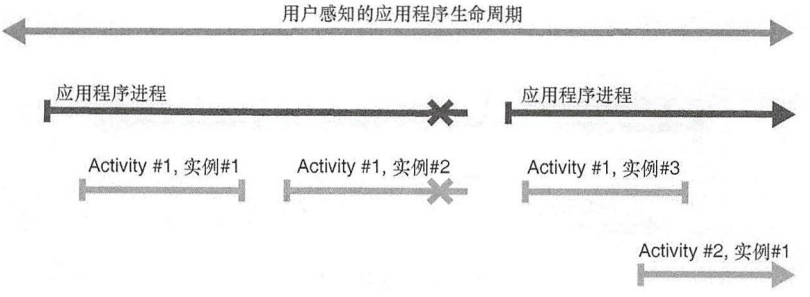


图 3.5　Android 应用程序生命周期

仅仅因为 Android 需要内存就丢失用户的状态，这是不能容忍的。用户希望有连

续、不间断的体验：他期望回到离开时状态相同的应用程序中，即使他是在几天前离开的。图 3.5 顶部的线段是用户感知的应用程序生命周期为这个期望的建模。

此外，应用程序的进程不可能持续几天。其他应用程序会进进出出，最终还会需要内存。对状态来说，唯一安全的地方是持久性存储，如文件系统、网络等。

Web 应用程序开发人员，尤其是那些使用 J2EE 的开发人员，会觉得图 3.5 似曾相识。Android 应用程序就是 Web 应用程序。

虽然桌面应用程序开发人员会仔细考虑应用程序的启动和停止，但 Web 应用程序开发人员不会这么做。Web 应用程序何时开始运行这个问题几乎没有任何意义，至少在涉及用户时是这样的。

这种类比可以扩展。将 Activity 组件是描述为具有自己生命周期、负责单个 UI 页面工作流程的对象，几乎和对 servlet 的描述完全相同。Android 服务组件类似于 J2EE 的会话 bean。即使.apk 文件（应用程序 bundle）强烈暗示.war 文件用于绑定 Java Web 应用程序。Android 应用程序是独立组件的集合，是在配置文件中声明，并在一个容器中运行的。

注意

将 Web 框架的演变与 Android 框架的演变进行比较是非常有趣的。视图、控制器和最终呈现的分离和用于将网络请求排队机制的改进，以及测试策略的进步都与 Web 应用程序开发有相似之处。

3.3 Android 进程

除非有其他具体的安排，否则应用程序代码将在一个标准线程上运行。这个线程称为主线程，有时称为 UI 线程。然而，大多数重要的 Android 应用程序都使用多个线程。

3.3.1 应用程序启动

每个 Android 应用程序都是 Zygote 这个原型应用程序的复制品。Zygote 的一个实例将作为启动系统的一部分。它自己初始化，预加载 Android 的大部分框架，然后等待连接到套接字。

当系统需要创建一个新的应用程序时，它就会连接到 Zygote 套接字，并发送一个小数据包的描述要启动的应用程序。Zygote 通过复制自己来创建一个新的内核级进程。

这个新进程很有趣，它和其父进程 Zygote 通过一个名为**写时复制**（copy-on-write）的模式共享内存。由于这两个进程使用相同的内存，因此子进程的启动几乎是瞬时的。内核不需要为新进程分配大量内存，启动也不需要再次加载 Android 框架库，它们都已经装入 Zygote。

写时复制意味着，如果子应用程序尝试修改共享空间中的值，那么内核就会分配新内存，并将这个页面复制到子进程写入的地方。新的子进程永远不会影响 Zygote 的内存。即使子进程要在每个内存页上写入（这永远不会发生），分配新内存所需的成本也会在应用程序的生命周期中逐步摊销，而不是初始成本。

图 3.6 所示的是 Zygote 的初始内存分配。Zygote 通过一个页表查看内存，这个页表将其内存引用映射到物理内存页上。

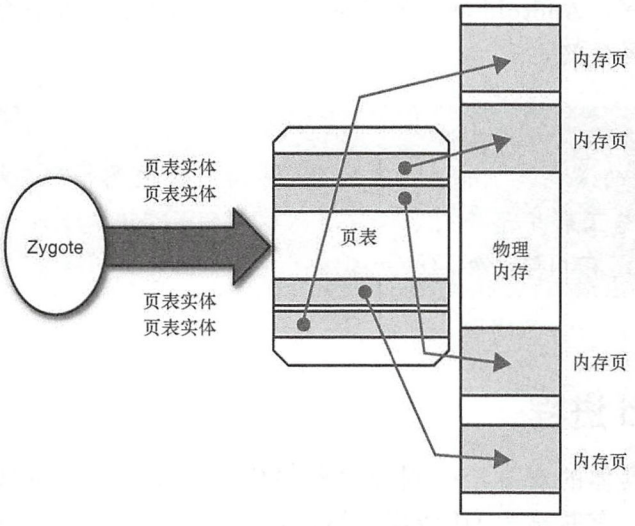


图 3.6　Zygote 内存

图 3.7 所示的是新应用程序中新进程的创建。新应用程序有自己的页表，大部分新页表都只是 Zygote 页表的备份。它指向物理内存中相同的页，只有新应用程序为自身目的而使用的页不会被共享。

图 3.8 所示的是当新应用程序尝试修改一个内存值时会有什么情况发生。由于页标记为页表中的写时复制，因此写入尝试会触发一个复制。页被复制，页表被调整以

指向新备份。当写入完成时，它只会修改新应用程序的内存，而不是 Zygote 的。

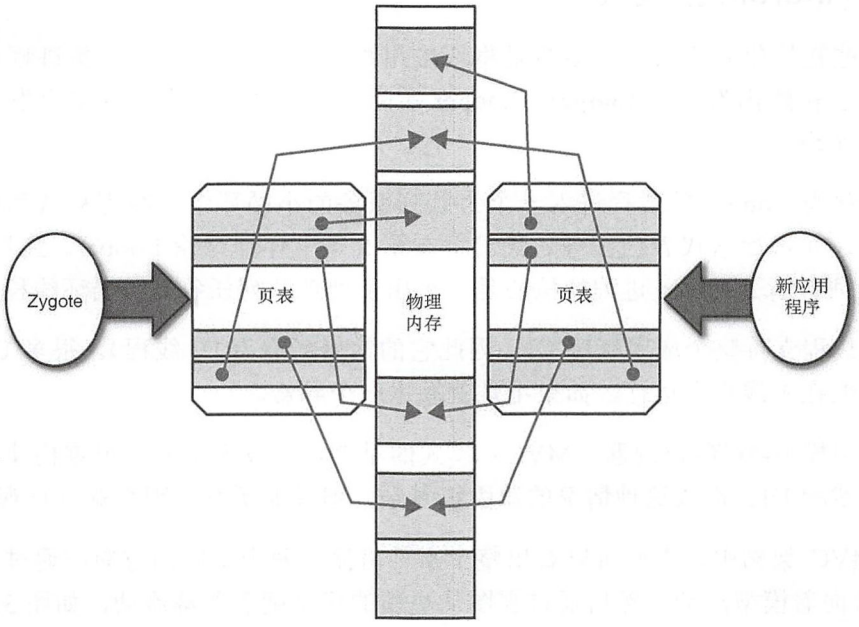


图 3.7 一个新应用程序共享 Zygote 的大部分内存

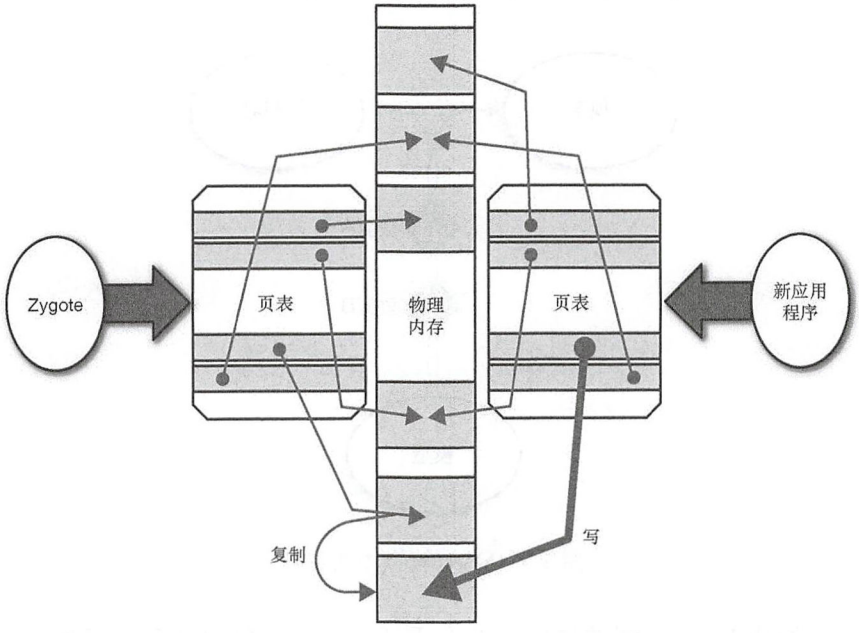


图 3.8 写时复制

3.3.2　Android 主线程

在一些初始化完成后（大多数是将新应用程序加载到内存中），新进程的主线程就会将自己初始化为一个 **Looper**。Looper 实际上是第 2 章中介绍的安全发布习惯用法的一种实现。

初始化为 Looper 的线程进入一个为队列服务的小循环中。线程从队列中删除下一个条目，并执行它代表的任务。我们将在第 5 章中详细讨论 Looper。到目前为止，我们可以理解新 Android 进程的核心是一个用于处理队列任务的小循环线程。

这个线程支持整个应用程序 UI（因此它的常用名称为 UI 线程）。很多 UI 方法都验证了它们在主线程上运行，如果不是就抛出一个异常。

在采用模型-视图-控制器（MVC）模式的早期，开发人员发现很难构建一个没有死锁的多线程 UI。造成这种情况的原因很复杂，但并非所有原因都难以直观理解。

在 MVC 架构中，人们可以看出整个事件将流入两个不同的方向：通过控制器从屏幕/键盘向着模型流动，然后通过视图从更新的模型朝着屏幕流动，如图 3.9 所示。如果有需要从处理入站和出站事件中捕获资源，这些资源很可能在处理不同方向的过程中以不同的顺序被捕获，这导致死锁几乎无法避免。

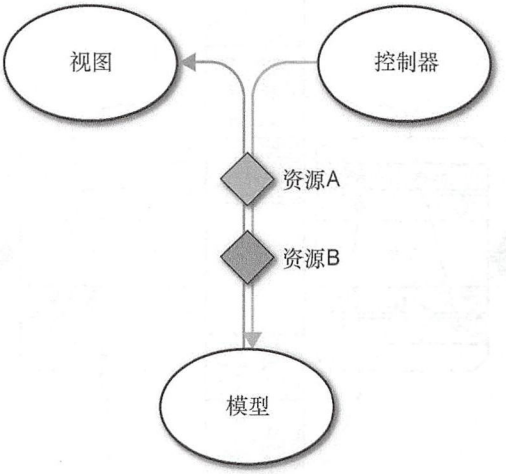


图 3.9　MVC 模型中的死锁

还有其他制作 UI 单线程的原因。两个重要的原因是简单性和原子性。

单线程 UI 很简单，因为它们在一个线程上运行。在单线程上运行的代码中引入锁、信号量、同步或线程安全完全没有必要。如果最多只有一个线程可以访问对象状态，那么这个规则已经满足了。

单线程 UI 的第二个方便之处：事件是原子的。当主线程从其队列中删除一个任务并开始执行它时，该任务就完全控制了 UI。例如，如果任务涉及在屏幕左下角激活按钮、在某个视图中替换图标，那么就会因为某种竞争条件显示新图标并激活按钮，而不是菜单项。在完成当前任务前，它会控制 UI 中所有的修改。从 UI 的角度来看，在队列中将它删除和它释放线程来处理下一个任务之间发生的一切，都是原子的。

当然，这也是单线程 UI 问题。Android 将帧速率的最低值定为 60fps。这意味着主线程至少每隔 17ms 就需要处理重绘任务。即使是本地文件系统，这也需要大量的 CPU 时间，而不是 I/O 时间。

因此，构建坚固、令人愉悦的 Android 的主要难点就变得显而易见了：

- 通过 UI 显示的应用程序状态必须对主线程可见；
- 通过 UI 显示的应用程序状态必须保存在持久存储器中（I/O）；
- I/O 必须在主线程上执行。当然，这包含读取和写入文件、数据库和网络。

3.4 小结

Android 应用程序不是一个普通的桌面应用程序，它更像一个网络应用程序——由一些类似于 servlet、控制器和网络开发人员所熟悉的数据访问层构建的。用户对应用程序生命周期的感知既与支持它的进程无关，也与组成它的组件无关。Android 创建了进程和组件，并可在由于其他目的需要资源时销毁它们。

Android 应用程序是驱动型的，除非开发人员做出特定的安排，否则就会由主线程驱动。这个线程就是 Looper，它是一个处理队列任务的紧密循环。它处理完一个任务后才会开始下一个任务。这很方便，因为它使任务原子化。然而，这可能也是有问题的。当一个任务涉及 I/O 或者由于某种原因需要大量时间来完成时，它就会关闭用户可见的 UI。

版权所有，严禁传播，违者自负法律责任！

第 4 章
AsyncTask 和 Loader

尽量理解作者想写什么，同时也不要责备他没能写出他根本就不想写的东西。

John Updike，《Picked-Up Pieces》

AsyncTask 是学习 Android 的第一个平台，是大多数开发人员会引入的第一个并发工具。对某些工作来说，它是一种聪明而有力的工具。然而，还未发现其他工具的开发人员经常在完全不合适的情况下使用它。和 5 年前相比，使用 AsyncTasks 最佳操作（和完全反对使用它的偏见）的社区知识变得更坚定且更广泛了。然而，在 StackOverflow Android 群组中发布包含“线程”这个词的问题几乎肯定都会导致一个涉及 AsyncTask 解决方案的回应——它是否适合。

4.1 AsyncTask 体系

从概念上来说，一个 AsyncTask 非常简单。它只是在另一个线程上执行一段代码的方式，如清单 4.1 所示。

清单 4.1 数据库查询

```
// Things that happen before the db query…

Cursor c = getContentResolver().query(
    DataProvider.URI,
    REQ_COLS,
    COL_LAST_NAME + "=?",
    new String[] { userName },
    COL_LAST_NAME + " ASC" );
```

版权所有，严禁传播，违者自负法律责任！


```
// Things that happen after the db query…
```

虽然这段代码不太好看，但它完全正确，并且完成了它应该做的工作。唯一的问题是因为数据库查询容易花费太多的时间，所以它不应该在主线程中执行。必要的修改似乎很少，并且似乎与从数据库中获取数据的工作毫不相关，因此应该可以在代码中进行微小的修改来完成它。如果能用某种理想化的计算机语言编写类似于清单 4.2 所示的代码，那就太好了。

清单 4.2 理想的背景化

```
// Things that happen before the db query…

Cursor c;
inBackground {
    c = getContentResolver().query(
        DataProvider.URI,
        REQ_COLS,
        COL_LAST_NAME + "=?",
        new String[] { userName },
        COL_LAST_NAME + " ASC");
}

// Things that happen after the db query…
```

可悲的是，Java 中没有 inBackground。

然而，这个想法并非一无是处。一些语言（例如 Swift 和 Scala）支持类似的结构。在这些语言中，inBackground 后面的代码块封装在一个名为**闭包**（closure）的语言结构中。

Java 8 在新的 lambda 表达式语法中支持闭包。从目前未命名的 Android N 开始，Android 的 Java 也将支持它们。假设清单 4.2 是未来的 Android 编译器可以编译的代码。闭包真的可以解决问题吗？这些就足够了吗？

在支持闭包的语言中，inBackground 不需要成为一个关键字。相反，它可以是对方法的调用，这个方法只有一个参数（闭包），并在另一个线程上执行这个闭包。清单 4.2 可以重写为清单 4.3。

版权所有，严禁传播，违者自负法律责任！

清单 4.3　将一个闭包传递给一个函数

```
// Things that happen before the db query…

inBackground(
    closure (localUserName = userName) {
        getContentResolver().query(
            DataProvider.URI,
            REQ_COLS,
            COL_LAST_NAME + "=?",
            new String[] { localUserName },
            COL_LAST_NAME + " ASC" );
    });

// State of the query is undefined, here…
```

显然，这仍然不是 Java。大致来说，inBackground 方法中的参数是对代码块的引用，而不是执行它的结果。在这个清单中，它是由创建的关键字 closure 来表示的。

inBackground 方法在另一个线程上运行闭包。特别是，可以想象它使用的是第 3 章描述的安全发布模式。它将闭包推入一个通过后台线程检索执行的队列中。

然而，这里还有几个未了结的部分。对闭包中的代码（封装的代码）有一些限制，必须使其与清单 4.1 所示的相应代码表现一致。

首先（至少在这个假想的语言中），闭包中的代码不能使用关键字 return、break 和 continue。这些关键字在闭包中的含义完全不同。例如，如果清单 4.1 包含一条 return 语句，那么执行它就会将控制权返回给调用者。如果这条 return 语句封装在闭包中并在后台线程中执行，那么它几乎肯定不会做同样的事情。

另一个限制是：如果闭包中的代码抛出一个异常，那么这个异常不会突然终止调用 inBackground 方法的代码。正如 return 语句中的情况所示，如果假定的 inBackground 方法在不同的线程上运行闭包，那么这个异常就会展开后台线程堆栈，而不是调用线程堆栈。

更严格的限制是，在封闭代码中使用的任何先决条件（例如变量等）都必须在闭包中有可用的值。当在完全不同的时间和环境中执行时，如果闭包的行为和原来的一样，那么闭包中使用的所有变量都必须具有闭包创建时所具有的值。

版权所有，严禁传播，违者自负法律责任！

假设常规的 Java 命名约定，封闭代码中唯一不表示常量的符号是 userName。假设清单 4.3 使用创建的 closure 关键字，将它赋给一个新的局部常量 localUserName。显然，如果封闭代码涉及很多变量，给其中的任何一个变量赋值，或者更糟糕的是，依赖于周围代码的副作用，那么事情会变得更复杂。

使用 Java 开发的人会意识到，现在可能做一些与在 Java 中类似的事情：使用匿名类。清单 4.3 可以用可编译的 Java 重写，如清单 4.4 所示。

清单 4.4 数据库查询

```
// Things that happen before the query…

final String localUserName = userName;

inBackground(
    new Runnable () {
        @Override public void run() {
            getContentResolver().query(
                DataProvider.URI,
                REQ_COLS,
                COL_LAST_NAME + "=?",
                new String[]{localUserName},
                COL_LAST_NAME + " ASC");
        } });

// State of the query is undefined, here…
```

Java 从创建起就一直可以模仿闭包行为。正如清单 4.4 所示，它的表达式相当冗长和严格。

现在这些限制也开始增加了。尝试在代码中表达“在不同的线程上运行这个代码段”变得非常困难。即使曾经作为匿名类实现，闭包无疑也是有用的工具。然而，很清楚的一点是，它们不会支持在另一个线程上执行任意未经修改的代码。

还有一个问题，它是最明显、最麻烦的。不管怎样，后置条件（由于执行封闭代码所引起的状态改变）必须返回给调用者。在清单 4.4 中，游标 c 必须可被跟随 inBackground 调用的代码来使用。

这个练习的关键是获取一段异步执行的代码。最明显的直接后果是，封闭代码

版权所有，严禁传播，违者自负法律责任！

不再严格地在闭包块后的第一行代码之前执行。在查询后以及获取游标后产生的代码不再这样做了。

AsyncTask 提供了这个问题的解决方案。

注意

事实上，在写本书时，引入到“N”版 Android Java 中的 Java 8 闭包完全不影响我们的讨论。不管是表示匿名类，还是实现新的 invokedynamic 指令闭包，都不足以将一个代码块移动到另一个线程中。

4.1.1　AsyncTask 基础

AsyncTask 的核心是上一节体系的延伸。它是受 Java 的 SwingWorker 的启发，其目标与之前的清单（将一段代码移动到后台线程中）完全相同，受到的约束也完全相同。它唯一的目的是在后台线程上运行一个代码段。理想情况下，这段代码是从其当前位置剪切并粘贴到任务中的，而剪切代码工作由一个执行任务的调用来代替。

从体系结构上来说，AsyncTask 类是经典的设计模式——类型安全模板的实现。定义它有一个抽象的方法，即模板方法 doInBackground。作为典型的模板模式，AsyncTask 类是抽象的，使用它的唯一方法是创建一个子类。为了在后台线程上执行代码，要创建一个 AsyncTask 的子类并将代码粘贴到这个子类的 doInBackground 方法的实现中。

AsyncTask 中封装代码的执行步骤如图 4.1 所示。

1）创建任务的新实例。一个任务实例只能使用一次，再次尝试运行会导致它抛出 IllegalException。

2）调用新实例的 execute 方法传递参数。

3）在调用者的线程上调用任务的 onPreExecute 方法。如果设置对所有任务的执行都是相同的，那么这个设置属于子类的 onPreExecute 实现。

4）任务的 doInBackground 方法是在执行器上调度的。传递给任务 execute 方法（步骤#1）的参数发布到执行器线程中，并称为 doInBackground 参数。

5）doInBackground 方法执行并完成。在执行过程中，调用任务的 cancel 方

版权所有，严禁传播，违者自负法律责任！

法会将任务标记为取消。在只有一个标记的情况下，取消并不一定会中断任务或者导致任务立即终止。然而，抛出到 doInBackground 方法中的异常会立即终止整个任务，并且后续的步骤都不会执行。

6）一个完成方法会在主线程上调度。如果在主线程恢复任务前，任务的 cancel 方法被调用，那么会使用 doInBackground 的返回值调用 onCancel。如果 cancel 没有被调用，那么会使用返回值调用 onPostExecute。

7）这个任务现在完成了并且应该可以释放了，这样它就能作为垃圾回收了。

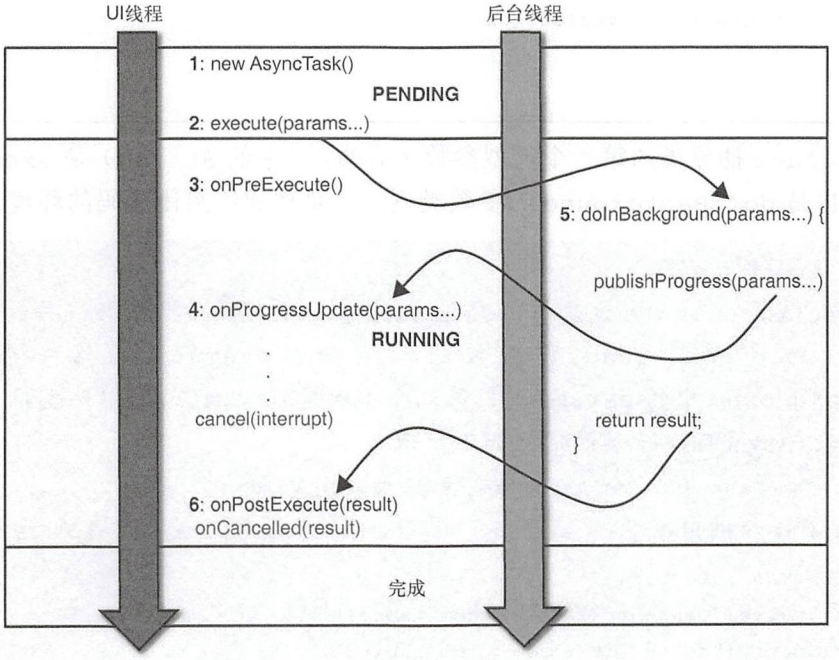


图 4.1 异步任务

正如清单 4.4 所示，在后台线程上执行的代码必须将自己的环境传递给它，并且必须能够返回其计算结果。AsyncTask 类通过使用 doInBackground 方法的参数和返回值明确这一点。它使用 Java 泛型使这些值成为类型安全的。

清单 4.5 所示的是以清单 4.1 中的代码作为 AsyncTask 实现。

清单 4.5 一个简单的异步任务

```
private class GetNamesTask extends AsyncTask<String, Void, Cursor> {
    @Override protected Cursor doInBackground(String… userName) {
        return getContentResolver().query(
```

版权所有，严禁传播，违者自负法律责任！


```
                DataProvider.URI,
                REQ_COLS,
                COL_LAST_NAME + "=?",
                new String[] { userName[0] },
                COL_LAST_NAME + " ASC");
        }

        @Override
        public onPostExecute(Cursor cursor) {
            onCursorReceived(cursor);
        }
    }
```

AsyncTask 抽象类的第一个泛型参数（清单 4.5 中的 String）是 execute 的参数类型，也是 doInBackground 的参数类型。这是传递给封闭代码的环境。

注意

AsyncTask execute 方法使用 Java **varargs**。它接收多个参数，并最终将它们传递给 doInBackground 作为数组。例如，execute 的第一个参数是 doInBackground 中的 param[0]、第二个参数是 param[1]，以此类推。

例如，AsyncTask 子类的定义如下所示：

```
class SomeTask extends AsyncTask<Uri, Void, Void>
```

可能会这样调用：

```
new SomeTask.execute(
   Uri.parse("content://sample.com/table1")
   Uri.parse("content://sample.com/table2"));
```

这种情况下，在 doInBackground 方法中声明为：

```
public void doInBackground(Uri… uris)
```

uris[0]是 uri content://sample.com/table1，uris[1]是 uri content://sample.com/table2。

唯一的约束是所有参数都必须是形参类型或者其子类型。例如，尝试使用一个 String 和一个 Uri 调用 execute 方法：

```
new SomeTask.execute(
    "content://sample.com/table1"
    Uri.parse("content://sample.com/table2"));
```

这可能会导致编译器错误。

版权所有，严禁传播，违者自负法律责任！

清单 4.5 还演示了将后台计算的结果返回给调用代码的 AsyncTask 解决方案。调用 doInBackground 的返回传递给另一个方法 onPostExecute。onPostExecute 在主线程上运行，并且保证在 doInBackground 完成后执行。最后，只有在后台代码已经运行后，才有空间放置可以执行的代码。第三个泛型参数（清单 4.6 中的 cursor）是 doInBackground 返回值的类型，因此也是 onPostExecute 的参数类型。

清单 4.6 执行一个 AsyncTask

```
    // Things that happen before the db query…

    new GetNamesTask().execute(userName);

    // …
}

void onCursorReceived(Cursor cursor) {
    // Things that happen after the db query…
}
```

注意

AsyncTask 的第二个泛型参数控制着由方法 publishProgress（在 doInBackground 中，由实现代码在工作者线程上调用）和 onProgressUpdate（作为响应，调用回主线程）实现的进度报告的参数类型。

这种机制很实用，但不会对 AsyncTask 有太大影响，因此不在这里讨论它。

最初按照程序顺序在数据库中查询后面的代码，现在是从任务调用的新线程。之前，程序顺序保证这段代码在查询后执行。现在，AsyncTask 约定保证它在查询后执行。

4.1.2 AsyncTask 的执行

AsyncTask 的准确实现随着时间而改变。回顾实际代码（不管是现在的还是历史的）会揭示一些与官方文档不一致的事实。

文档声称在 API level 4 Donut 前，AsyncTask 机制都是单线程的。AsyncTask（就像它的近亲 AsyncQueryHandler 一样）在 Android 的老版本中完全有可能在一个单后台线程上运行。然而，早在 API level 3 Cupcake 上，它们就已经像现在这样了，在

版权所有，严禁传播，违者自负法律责任！

Java 线程池执行器 THREAD_POOL_EXECUTOR 上执行。直到最近，不管是何种设备，这个执行器都是在 1～10 个线程上使用的。

在 2011 年临近 API level 11 Honeycomb 发布时，这个策略发生了巨大改变。AsyncTask 的新实现并没有将任务直接提交给可能会乱序执行的执行器，而是引入了一个新的执行器 SERIAL_EXECUTOR。这个新执行器是前面有一个队列的旧执行器。提交给 SERIAL_EXECUTOR 执行的任务先推入队列中，它们一直在那里等待，直到之前提交的任务完成执行为止。轮到某个任务时，它就会在执行器上执行，这个执行器本质上等于 Cupcake 中使用的执行器。然而由于队列，所以任务是按照提交顺序执行的，每个任务运行完成后下一个任务才能开始执行。

毫无疑问，就像 Android 文档所说，这种改变“避免了由并发执行导致的常见应用程序错误”。可以想象一下，由于尝试将某些内容保存到不存在的数据库中或者写入一个尚未打开的文件中而导致很少见的崩溃时，开发人员焦头烂额的样子。现在这种事不太可能会发生了。

此外，AsyncTask 不再并发执行了！然而，它有一种返回并发执行的方式。在正常的 AsyncTask 执行被序列化的同时，框架引入了一种新方法 executeOnExecutor。新方法的第一个参数将提交 AsyncTask 的执行器。由于空线程池执行器 THREAD_POOL_EXECUTOR 是 AsyncTask 类中的公有常量，因此想要并发执行的开发人员可以通过如下调用绕过队列：

```
task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,…)
```

要注意的是，使用这种方法，还可以将 AsyncTask 加入到应用程序特定执行器的队列中。例如，发现默认的 AsyncTask 串行执行模式有吸引力但必须偶尔长时间运行后台任务的开发人员，可以考虑创建只用于并发执行较慢任务的第三个执行器 LARGE_TASK_EXECUTOR。这种架构（为不同类型的任务创建分层优先级的方案）非常有效。

任务现在按顺序运行这个事实并不意味着它们都在一个线程上运行！这是官方文档的第二个偏差。执行器完全有可能为所有任务使用一个线程。如前所述，API Level 19 之前的 Android 版本用单个线程初始化 AsyncTask 执行器线程池。不管是因为偶然还是因为队列，如果每次只有一个任务提交给执行器，那么永远不会在其线程池中增加多个线程。这种情况下，单个线程可以运行所有任务。

2013 年，对于 API Level19 来说，线程池的核心大小修改为 CPU 数+1。池的最大值修改为 CPU 数的 2 倍+1。从那时开始，执行器的线程池可能至少有两个线程，并且

版权所有，严禁传播，违者自负法律责任！

可以有多个线程。

这很重要，因为它意味着 AsyncTasks 的 doInBackground 方法在不同步时，无法安全通信。如果能保证所有 doInBackground 方法在同一个线程上运行，那么清单 4.7 就是正确的。由于没有这样的保证，因此这段代码是不正确的。

清单 4.7 不正确的跨任务通信

```
static boolean mFlag;

private class IncorrectTask extends AsyncTask<Void, Void, Void> {
    @Override protected Void doInBackground(Void… empty) {
        if (!mFlag) { // !!! Incorrect
            mFlag = true;
            // …
        }
        // …
        return null;
    }
}
```

4.1.3 AsyncTask 的完成

AsyncTask 框架不支持从 doInBackground 方法内部抛出的异常。如果 doInBackground 中的代码突然终止，整个任务生命周期也会突然终止，onPostExecute 和 onCancelled 都不会被调用。这几乎肯定是一个程序错误。doInBackground 方法应该总是可以正常退出。

当 doInBackground 完成时，AsyncTask 框架就会通过在主线程上调度执行 onCancelled 或 onPostExecute 来完成任务的生命周期。通常调用的是 onPostExecute 方法。然而，如果任务的 cancel 方法是在任务开始（调用它的 execute 方法）和框架决定调用哪个完成方法时调用的，就会调用 onCancelled 方法。

要注意的是，AsyncTask 约定不保证 onPostExecute 或 onCancelled 的完成方法会在调用 execute 方法的线程上运行。它只能保证该方法在主线程上运行。如果 execute 是在另一个线程上调用的（例如，在另一个 AsyncTask 的 doInBackground 方法上），那么就会在调用 execute 方法的线程上调用 onPreExecute。然而，onPostExecute 方法将在主线程上运行。在这种情况下，假设 onPreExecute 和

非卖品，仅供非商业用途或交流学习使用

onPostExecute 在同一个线程上运行是无效的，并且关于并发访问可变状态的所有规则都适用。

调用任务的 onCancelled 方法并不能证明由 doInBackground 方法可观察到对 cancel 方法的调用。甚至有几种方式可以使在 doInBackground 正常完成后调用 onCancelled。其中最简单的是对 cancel 的调用发生在 doInBackground 完成和在主线程调度完成方法期间。

然而，另一种可能发生的原因是 cancel 方法很有礼貌。默认的实现不会强迫 doInBackground 方法中止。它只设置一个表明任务已经取消的标志，如果 cancel 的布尔参数设置为 true，那么就使用 Java 的线程中断机制请求运行任务的线程停止这样执行。doInBackground 的实现才能决定是否注意到这个请求并付诸行动。

如果任务等待一个阻塞操作完成，当将调用 cancel 的参数设置为 true 时，线程就会中断并且操作中止。当中断时，大多数 I/O 操作和所有低级的线程阻塞操作都会突然终止。I/O 操作抛出一个 InterruptedIOException，而线程阻塞操作抛出一个 InterruptedException。不管是哪种情况，正确编写的 doInBackground 方法必须捕获一个异常，并返回一个不完整的值或者失败标志。当然，即使代码没有被阻塞且线程中断发生在其他地方，使能 cancel(true) 中断的任务也必须优雅地处理它们。

此外，如果 doInBackground 中的计算是 CPU 密集型的（也许是位图转化或者将 π 的值计算到百万位数），它必须周期性地检查是否被中断。这种检查是通过调用 AsyncTask 方法 isCancelled 完成的。

注意

不要掩饰中断！

中断一个线程会在 Thread 对象中设置一个标志。Thread.interrupted 方法会返回这个标志值并清除它！作为响应中断而突然终止的大多数阻塞操作，也会在它们抛出前清除这个标志。Thread. currentThread().isInterrupted 方法是在不改变中断状态而发现它的唯一方式。

设计可中断任务时，中断标志的状态是 API 的一部分。如果调用者需要知道中断，一定要在使用 Thread.interrupted 测试其状态或者捕获一个中断异常后，使用 Thread. currentThread().interrupt 方法重新设置该标志。

让 doInBackground 方法成为可取消的可能是个重大的挑战，可能需要让架构的

非卖品，仅供非商业用途或交流学习使用

修改远离任务本身。如清单 4.8 所示，它是一个与前述示例非常相似的任务。

清单 4.8 不正确的跨任务通信

```
private class DbInsert extends AsyncTask<ContentValues, Void, Integer> {
    @Override protected Void doInBackground(ContentValues… values) {
        return Integer.valueOf(
            getContentResolver().bulkInsert(DataProvider.URI, values))
    }
}
```

这个任务是进行一次潜在的大插入，而不是查询。它有可能需要几秒钟的时间才能完成。然而，SQLite 操作是不可中断的。大多数 SQLite 实现都是可移植 C 代码的，它完全没有注意到 Java 的线程中断标志。

那么，它是否可以让这个任务成为可取消的？在某些情况下，这是可能的。例如，可能需要修改供应商的架构才能让任务成为可取消的。这个示例很有启发性。

要牢记的是，对 ContentResolver.bulkInsert 的调用被由 URL: DataProvider.URI 授权的内容提供者代理。对 ContentResolver. bulkInsert 的调用由提供者的 bulkInsert 方法来处理。如果清单 4.8 中的异步任务是可中断的，它就必须支持中断的方法。清单 4.9 说明了这一点。

清单 4.9 可中断的批量插入

```
@Override
public int bulkInsert(Uri uri, ContentValues[] rows) {
    SQLiteDatabase db = getDb();
    db.beginTransaction();
    try {
        int inserted = 0;
        for (int len = rows.length; inserted < len; inserted++) {
            db.insert(Db.TABLE, null, rows[inserted]);
            if (Thread.interrupted()) { break; } // not recommended :-(
        }
        db.setTransactionSuccessful();
    }
    finally { db.endTransaction(); }
    return inserted;
}
```

非卖品，仅供非商业用途或交流学习使用

这种把戏很脆弱，是我们绝对**不推荐**的。例如，只有当 bulkInsert 方法和 doInBackground 方法在同一个后台线程上运行时，它才有效。如清单 4.9 所示，如果内容提供者在单独的进程中运行，那么它就不会注意或者接受中断。一种更好的解决方案是让 DbInsert 将记录分成多份，对提供者的独立 insert 调用，并检查这些调用之间的 isCancelled 方法。

若想让一个 AsyncTask 成为可取消的，要充分依赖于任务本身设计者没有控制权的代码。

4.2　使用 AsyncTask

理解 AsyncTask 的实现后，现在让我们探讨如何有效地在应用程序中使用它们。这是一个简单的工具！什么可能会出错？

4.2.1　AsyncTask：被认为是危险的

可悲的是，有很多可能会出错的事情。AsyncTask 的错误可分为两大类：并发错误和生命周期错误。

1. 出错：并发

并发错误违背了 Java 并发规则，如清单 4.10 所示。

清单 4.10　有并发错误的 AsyncTask

```
@Override
public void onClick(View v) {
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... args) {
            String msg = textView.getText(); // !!!
            textView.setText("");            // !!!
            network.post(msg);
            return null;
        }
    }.execute();
}
```

非卖品，仅供非商业用途或交流学习使用

doInBackground 方法的关键是它在不同的线程上运行！细心的读者会注意到，对 textView.getText 的调用是对其他线程不可见对象（在这种情况下，主线程拥有 View 对象）的同步引用。这段代码是不正确的。

幸运的是，这段代码会在运行时以非常明显的方式显示失败。Android 视图框架中的大多数方法都会验证它们是从主线程中调用的，如果不是就会显示失败。在 AsyncTask 调用 setText 时会产生一个 ViewRootImpl$CalledFromWrong ThreadException。但要注意的是，调用 getText 同样是不正确的，但不会产生异常。

在第 8 章讨论的静态分析工具可以在运行前识别出这类问题。在这些工具中，Android Studio 的最新版本支持新的 Android 注释@UiThread、@MainThread、@WorkerThread 和@BinderThread。应用于方法中的这些注释有助于静态分析工具报告不正确的代码。

清单 4.10 展示了由于 doInBackground 方法中变量作用域而引发的并发问题。清单 4.11 所示的是相关的问题。

清单 4.11 有并发错误的另一个 AsyncTask

```
@Override
public void onClick(View v) {
    new AsyncTask<TextView, Void, Void>() {
        @Override
        protected Void doInBackground(TextView... args) {
            TextView view = args[0]
            String msg = view.getText(); // !!!
            view.setText("");            // !!!
            network.post(msg);
            return null;
        }
    }.execute(textView);
}
```

将可变值作为参数传递给 doInBackground 并没有任何帮助！实际上，它只会使问题更难以确定。局部变量 view、方法参数 args[0]和类成员 textView 都指向同一个对象。由于这些引用在多个线程中可见，因此代码是不正确的。

清单 4.11 将在运行时失败，就像清单 4.10 所示的代码那样。清单 4.12 再次说明了这个问题，但它会以一种不太可预测的方式来显示失败。

非卖品，仅供非商业用途或交流学习使用

清单 4.12　有并发错误的另一个 AsyncTask

```
@Override
public void updateHandler(List<String> strings) {
    new AsyncTask<List<String>, Void, Void>() {
        @Override
        protected Void doInBackground(List<String>... args) {
            List<String> s = args[0]
            for (int i = 0, n = s.size(), i < n; i++) {
                network.post(s.get(i));
            }
            return null;
        }
    }.execute(strings);
}
```

这里的问题是代码又一次将可变对象的引用泄露给了工作线程。updateHandler、strings 的参数在 doInBackground 称为 args[0]和 s。由于 strings 是作为参数传递给 updateHandler 可变数据结构引用的，因此不可能知道在程序其他位置有多少别的引用被使用。

注意

Collections.unmodifilable 不是一个解决方案。

将可变对象封装在不可变的视图中并不能解决问题。例如，Java 集合库方法 Collections.unmodifiableList 只创建了一个不可修改的列表视图。除非用不可变的封装器将对列表的所有引用封装起来，否则运行 doInBackground 方法的线程仍然持有对对象的引用，它可以在另一个线程上修改。

让代码正确的唯一方式是创建所有可变数据结构的本地备份，如清单 4.13 所示。在这种情况下，这只是意味着复制列表。如果列表的元素是可变的，那么它们也必须复制。

清单 4.13　一个问题不太大的 AsyncTask

```
@Override
public void updateHandler(List<String> strings) {
    new AsyncTask<List<String>, Void, Void>() {
```

非卖品，仅供非商业用途或交流学习使用

```
        @Override
        @WorkerThread
        protected Void doInBackground(List<String>... args) {
            List<String> s = args[0];
            for (int i = 0, n = s.size(), i < n; i++) {
                network.post(s.get(i));
            }
            return null;
        }
    }.execute(new ArrayList(strings));
}
```

综上所述，问题就是 Java 的块结构和线程结构并不总能协调一致。自动发生的事情有时很危险。在 AsyncTask 中正确并发需要注意细节。而且，并发并不是它的终点。

2. 出错：生命周期

使用 AsyncTask 最常见的错误是与 Android 组件生命周期相关的错误。清单 4.14 和清单 4.15 是显示这种错误的完美示例。这两个示例都尝试将数据发布到网络中，从而作为对用户输入的响应。考虑一下，代码为什么可能会失败？

假设用户将包含这段代码的应用程序安装到手机上，并且他在乘坐地铁失去信号时偶然调用了 AsyncTask。在没有连接的情况下，发送请求最终会超时。

合理的超时间隔可能是 60s。虽然很多移动应用程序都选择小得多的数字（10～15s）作为用户等待的时间，但对异步执行的任务来说，30～90s 可能是有意义的。

当连接超时后，它可能会使用某种后退算法进行重试。例如，清单 4.12 中的 post 方法可能会在 30s、90s 和 180s 后重试。在用户采取启动操作后，这个任务在整整 3min 内不停地工作。

回想一下图 3.4，它展示了一个活动的生命周期，以及它如何与应用程序生命周期的用户感知关联起来。图 4.2 所示的是同样的生命周期，但添加了由活动产生的 AsyncTask。

Android 框架控制着 Activity 的生命周期。有很多原因（屏幕旋转是最常引用的）会导致框架决定放弃一个 Activity 实例，稍后用另一个实例替代它。

然而，AsyncTask 的生命周期完全无法管理。图 4.2 演示了正在考虑的示例，（在

非卖品，仅供非商业用途或交流学习使用 严禁网络传播本PDF，违者责任自负！

生成它的 Activity 离去后）任务已经存在好几分钟了。

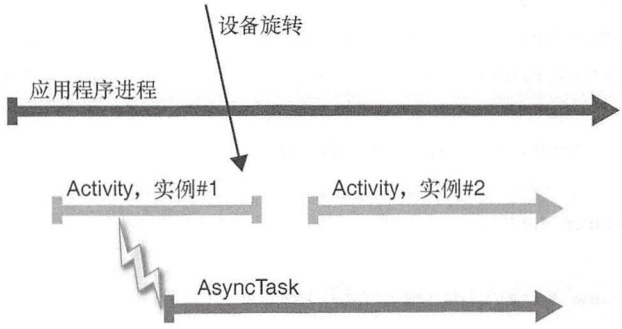


图 4.2　AsyncTask 的生命周期

要确切地理解这为什么是个问题，首先要对清单代码进行细微的重构。在清单 4.14 中，任务提交给命名内部类 `PostTask`，而不是以前使用的匿名类。尤其要注意的是，`PostTask` 在其构造函数中将 `Context` 作为其参数，并将对它的引用保存在创建的网络对象中。传递给 `PostTask` 的上下文就是当前的 Activity。

清单 4.14　有生命周期错误的 AsyncTask

```
private static class PostTask extends AsyncTask<String, Void, Void>() {
    private final Network network;

    public PostTask(Context ctxt) { this.network = new Network(ctxt); }

    @Override
    protected Void doInBackground(String... args) {
        network.post(args[0]);
        return null;
    }
}

    // …
    public void onClick(View v) {
        view.setText("");
        new PostTask(this).execute(textView.getText().toString);
    }
```

产生的问题如图 4.3 所示。

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

由于任务持有对 Activity 的引用，因此即使在 Android 框架已经创建了替代品后，Activity 也不能被垃圾回收。如图 4.3 所示，AsyncTask 挂起了好几分钟。在此期间，冗余的 Activity（一个非常大的对象），以及它引用的大多数对象（可能是整个冗余的视图层）都困在内存中。

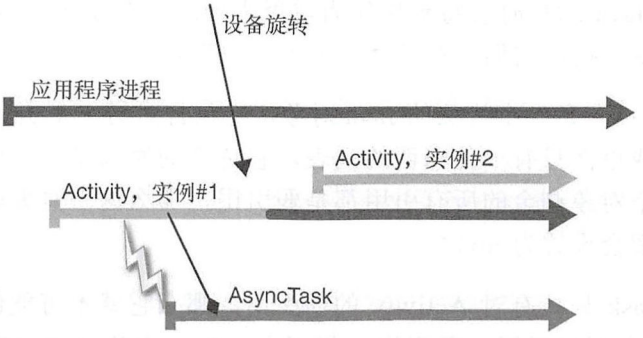


图 4.3 泄漏的 AsyncTask

要注意的是，作为非静态内部类（包含所有匿名类）的所有 AsyncTask 实现都存在这个问题！在 Java 中，在另一个类 A 中定义的任何非静态类 B 都持有对创建它的 A 实例的引用。例如，在清单 4.15 中，B1 和 B2 这两个类有几乎相同的实现。

清单 4.15 内部类实现

```
class A {
    private class B1 { }

    private static class B2 {
        private A parent;
        public B2(A parent) { this.parent = parent }
    }
}
```

AsyncTask 的每个匿名子类都在内存中持有创建它的 Activity，直到任务完成为止。

僵尸 Activity 是问题的另一个原因是：实际上任务可能尝试使用它持有的引用。注意，若 `AsyncTask.onPostExecute` 方法在主线程上运行，就可以原谅开发人员认为他已经找到通知用户任务完成的一种完美方式了。但他只是将通知视图的引用保存在任务中，并在 `onPostExecute` 中更新它。

图 4.3 说明了它不起作用的原因。在最好的情况下，`onPostExecute` 方法会更新

非卖品，仅供非商业用途或交流学习使用 严禁网络传播本PDF，违者责任自负！

一个不再连接到屏幕上的僵尸视图。在最坏情况下，由于 Activity 在其生命周期内一直运行并且调用了 `onDestroy` 方法，因此它是在不一致状态下的，并且会抛出异常而导致应用程序崩溃。

弱引用通常认为是这个问题的解决方案。经验丰富的 Java 开发人员会记得，当没有强引用时，在 Java 运行时会将对象作为垃圾来处理。在 Java 中几乎所有引用都是强引用：只要有变量指向它们，对象就永远不是垃圾。

`WeakReference` 是一个特殊的 Java 对象。它包含对另一个 Java 对象的引用。人们可能会把它看成里面只有一件东西的列表。它特殊的原因是：它引用的对象能被垃圾回收。如果一个对象剩余的所有引用都是弱引用，那么这个对象就会被垃圾回收，并且所有弱引用都会设置为 `null`。

如果 AsyncTask 只持有对 Activity 的弱引用，那么它就不可能创建僵尸进程。当对 Activity 的所有框架引用（硬引用）都不存在时，它就会被垃圾回收。稍后，当 AsyncTask 尝试指向 Activity 时，它会进行简单的检查，这样就可以发现对它的引用是空的，并采取适当的行为。这似乎是一个银弹。但事实并非如此，还有第 3 个原因。

注意

AsyncTask 几乎从来都不是网络操作的正确选择。

这好像是一个有力的声明。APP 商店中有很多应用程序，有些可能是 4 星和 5 星评级，这些应用程序都依靠 AsyncTask 的网络架构。

为网络操作选择一种更好的架构意味着有更少古怪的行为，能给你更多时间专注于愉悦用户。

最明显的原因是任务可能需要采取行动，而且可能无法这样操作。在前面提到的场景中，用户认为他已经将某些内容发布到网络上 3min 了，但其实发布失败了。如果 AsyncTask 不能报告错误，应用程序就会违背承诺。用户的发布没有成功，也没有任何可靠的方式通知他失败了。

第二个原因是成本。考虑一个相当复杂的 AsyncTask：可能会调用两个或者多个 REST 服务，并计算它接收数据的某种聚合。现在考虑一下，如果任务在其生命周期早期失去 Activity 会发生什么。它会进行昂贵的网络传输，然后加入 CPU 计算聚合，最后**将结果丢弃**。数据传输不会花费用户的现金，但计算确实要消耗电池的实际寿命。然而当工作完成时，没有放置结果的地方，结果只能被丢弃。真是太浪费了！

使用弱引用将 AsyncTask 嵌入 Activity 中，这个设计很糟糕的最后一个原因是它

严禁网络传播本PDF，违者责任自负！

真的是糟糕的设计。对僵尸活动的关注并不是需要解决的问题，它是糟糕架构的产物。长时间运行的异步任务根本不属于 Activity。Activity 是生命很短的对象。第 3 章将它们与 servlet 进行了比较。没有人会将业务逻辑放在 servlet 中。第 3 章还指出，只包含可见 Activity 的进程有资格进行进程收割。在只包含不可见 Activity 的应用程序中运行 AsyncTask 可以在没有警告的情况下被杀死，并且没有机会报告失败。

和 3 层网络服务一样，Android 应用程序应该将大量异步工作放入更合适的组件中。在 Android 中，该组件通常是一个 Service。Service 是第 6 章的主题。

4.2.2 使之正确

到目前为止，可以原谅认为 AsyncTask 永远不应该使用的读者，经常在 Android 开发者社区中听到这种反对观点。已经完成一两个 Android 应用程序的开发者之间的讨论通常包含一个心照不宣的、非特定的假设，即 AsyncTask 很糟糕。

正如软件开发中经常发生的一样，正是具体细节造成了这种差异。AsyncTask 不是万能的，然而，它们也不是一无是处的。了解它们何时有用以及何时没有用，会在评估未来热点时很有帮助。

AsyncTask 有自己的场所。下面是构建一些有弹性且实用的 AsyncTask 的方法。

1. 自发性任务

要取得自发性资格，AsyncTask 必须只有一个副作用：它不可以尝试再次同步并将信息（甚至是状态）返回给调用者。它还应该在相对短时间运行——不超过几秒钟。

在完成可能成功或者失败时进行检测和重试时，这种任务会非常有用。例如，创建并初始化一个文件就适合这种任务。如果应用程序需要的文件不存在，那么它就会启动一个创建该文件的任务。在未来的某个时间，当它再次发现需要这个文件时，文件可能就在那里。如果没有，它就开始执行另一项任务。

2. 可取消的任务

一个可取消的任务是停止处理响应 `cancel` 方法的任务。这种任务会在接收到取消通知时尽快终止它的 `doInBackground` 方法。当执行器释放它并且完成其生命周期时，它就会被垃圾回收，并且不会保留它持有引用的对象。

当然，一个可取消的任务需要被取消！当 Activity 开始一个可取消的 AsyncTask

严禁网络传播本PDF，违者责任自负！

时，必须记住它然后取消它，通常是响应 onPause 或 onStop 事件。

4.3　Loader 和 CursorLoader

CursorLoader 是一个恰当使用 AsyncTask 近乎完美的清单。它的基类 Loader 是异步数据加载的抽象。子类 AsyncTaskLoader 作为 AsyncTask 的抽象实现。CursorLoader 是 AsyncTaskLoader 的子类，它可能是本章开头问题的最好回答：从数据库中异步加载一个充满数据的游标。

《The Busy Coder's Guide to Android Development》（2009）的作者 Mark Murphy 曾经将加载器（Loader）称为“失败的抽象”。这是准确的评估。加载器并没有作为将数据加载到视图中的通用解决方案。虽然它不是成功的通用抽象，但其具体的派生类 CursorLoader 是一个流行且可靠的工具。

使用 CursorLoader 的目标是从数据库中获取数据。最初的解决方案看起来有点不可思议。它是看起来完全偏离方向的过程，包含如下 3 个步骤。

1）让加载管理器创建一个加载器。

2）在加载管理器请求时，给它一个加载器。

3）接收加载器返回的游标。

考虑一个在列表视图中显示数据查询结果的应用程序。框架 Activity 看起来和清单 4.16 很像。

清单 4.16　框架游标列表 Activity

```
public class LoaderActivity extends ListActivity {

    // ...

    @Override
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(
            this,
            R.layout.list_row,
            null,
```

严禁网络传播本PDF，违者责任自负！

```
            FROM,
            TO,
            0);

        setListAdapter(adapter);
    }

    // ...
}
```

SimpleCursorAdapter 构造函数的第三个参数是内容要在列表视图中显示的游标。在清单 4.16 中没有游标要显示，因此实际参数是 null。本章剩余部分的重点是用从数据库中异步获取且充满数据的游标代替这个值。

刚刚描述的第一步是让加载管理器启动一个加载器，这需要两行代码，如清单 4.17 所示。

清单 4.17 初始化加载器

```
public class LoaderActivity extends ListActivity {
    private static final int DATA_LOADER = -8954;

    // ...

    @Override
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(
            this,
            R.layout.list_row,
            null,
            FROM,
            TO,
            0);

        setListAdapter(adapter);

        Bundle args = getIntent().getExtras();
        getLoaderManager().initLoader(DATA_LOADER, args, this);
```

严禁网络传播本PDF，违者责任自负！

```
        }

        // ...
    }
```

调用 initLoader 第一个参数是初始化的加载器名字。它是一个整数：应用程序中任何唯一的整数都满足要求。

回想一下，Activity 的 onCreate 方法会多次调用，有时在 1s 会调用多次。每次调用 onCreate 时都创建一个新的加载器实例非常浪费。相反，当 initLoader 被调用时，它会检查给定名字的加载器是否已经存在。只有不存在时，它才会初始化并启动一个新加载器。

第二个参数是传递给加载器创建的参数。它们稍后会再次出现。

initLoader 的第三个参数是回调处理函数。在 Android 框架调用有生命周期事件的分形重复方式中，加载管理器会用表示加载器生命周期的事件调用回调处理函数。

将进程和异步协作者之间的交互表示为事件流是一种模式，这将成为本书剩余部分重要的架构工具。在这种情况下，有 3 个事件：onCreateLoader、onLoaderFinished 和 onLoaderReset。

要想接收这些事件，回调处理函数必须实现接口 LoaderManager.LoaderCallbacks<T>（声明这 3 种方法），其中 T 是加载器将要返回的对象类型。在这种情况下，该对象是一个游标。清单 4.18 说明了扩展 LoaderActivity 以接收加载器生命周期事件。

清单 4.18　接收加载器事件

```
public class LoaderActivity
    extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor>
{
    private static final int DATA_LOADER = -8954;

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) { }

    @Override
```

严禁网络传播本PDF，违者责任自负！

```
    public void onLoadFinished(Loader<Cursor> l, Cursor c) { }

    @Override
    public void onLoaderReset(Loader<Cursor> c) { }

    @Override
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(
            this,
            R.layout.list_row,
            null,
            FROM,
            TO,
            0);

        setListAdapter(adapter);

        Bundle args = getIntent().getExtras();
        getLoaderManager().initLoader(DATA_LOADER, args, this);
    }

    // ...

}
```

在调用 initLoader 后的某个时刻，加载管理器会调用回调处理函数（这个清单中是 Activity）来请求一个加载器。这是 3 个生命周期事件中的第一个。由于这里的目标是加载游标，因此 Activity 将提供一个 CursorLoader，如清单 4.19 所示。

清单 4.19 创建一个 Cursor Loader

```
public class LoaderActivity
    extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor>
{
    private static final int DATA_LOADER = -8954;

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
```

严禁网络传播本PDF，违者责任自负！

严禁网络传播本PDF，违者责任自负！


```
            return new CursorLoader(
                this,
                DataProvider.URI,
                new String[] {
                    DataProvider.Columns.ID,
                    DataProvider.Columns.FNAME,
                    DataProvider.Columns.LNAME},
                DataProvider.Columns.LNAME + "=?",
                new String[] { args.getString(DataProvider.Columns.LNAME) },
                DataProvider.Columns.LNAME + " DESC");
        }

        @Override
        public void onLoadFinished(Loader<Cursor> l, Cursor c) { }

        @Override
        public void onLoaderReset(Loader<Cursor> c) { }

        @Override
        protected void onCreate(Bundle state) {
            super.onCreate(state);

            SimpleCursorAdapter adapter = new SimpleCursorAdapter(
                this,
                R.layout.list_row,
                null,
                FROM,
                TO,
                0);

            setListAdapter(adapter);

            Bundle args = getIntent().getExtras();
            getLoaderManager().initLoader(DATA_LOADER, args, this);
        }

        // ...

    }
```

在新代码中有两件事情需要注意。首先，在初始化时传递给加载管理器作为第二个参数的包，在这里作为参数 `args` 出现。代码使用这个包中的一个值作为查询 `WHERE` 子句的参数。

然而，最重要的是 `CursorLoader` 的构造参数与数据库查询语句的参数几乎完全相同：URI、投射、限制、限制参数和排序。`CursorLoader` 的返回实例包含对所要运行查询的表示这个表示是完整且已被封装的。

加载管理器有了加载器后，就可以异步运行了。如前所述，`CursorLoader` 是从 `AsyncTaskLoader` 处继承的，它使用 `AsyncTask` 的新实例以在后台运行嵌入到加载器中的查询。图 4.4 演示了加载器的生命周期。

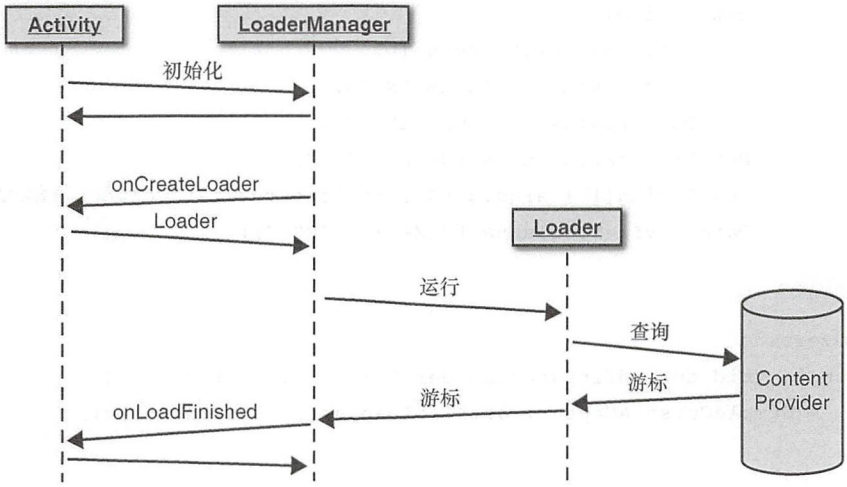


图 4.4 加载器的生命周期

Activity 初始化加载器，调用立即返回。稍后，加载管理器回调 Activity 来获取一个加载器。加载器创建 AsyncTask 子类的一个新实例并运行它来传递查询参数。

在后台线程上运行的 AsyncTask 可以查询数据库，并最终得到一个充满数据的游标。在 `onPostExecute` 方法中，它回调加载器中的完成方法来传递游标，而这个完成方法反过来又回调 Activity 的 `onLoadFinished` 方法。这就是第二个加载器生命周期事件。

清单 4.20 显示了完整实现的活动。

严禁网络传播本PDF，违者责任自负！

清单 4.20 完整的 Cursor Loader

```
public class LoaderActivity
    extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor>
{
    private static final int DATA_LOADER = -8954;

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        return new CursorLoader(
            this,
            DataProvider.URI,
            new String[] {
                DataProvider.Columns.ID,
                DataProvider.Columns.FNAME,
                DataProvider.Columns.LNAME},
            DataProvider.Columns.LNAME + "=?",
            new String[] { args.getString(DataProvider.Columns.LNAME) },
            DataProvider.Columns.LNAME + " DESC");
    }

    @Override
    public void onLoadFinished(Loader<Cursor> l, Cursor c) {
       ((SimpleCursorAdapter) getListAdapter()).swapCursor(c);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> c) {
       ((SimpleCursorAdapter) getListAdapter()).swapCursor(null);
    }

    @Override
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(
            this,
            R.layout.list_row,
            null,
```

```
                FROM,
                TO,
                0);

        setListAdapter(adapter);
        Bundle args = getIntent().getExtras();
        getLoaderManager().initLoader(DATA_LOADER, args, this);
    }

    // ...

}
```

当从加载器接收游标时，onLoadFinished 方法只是将新游标交换到 onCreate 中。游标的内容会突然出现在列表视图中。

还有一件事要注意，加载器确实符合前面提到的 AsyncTask 模式。它是可取消的。

虽然从代码中并不能明显看出这些，但加载管理器会观察加载器所连接 Activity 的生命周期事件。一旦确定相关的 Activity 正在被销毁，并且没有其他 Activity 实例立即代替它（也就是说，不仅是屏幕旋转或者类似的事件），它就会恰当地取消加载器。

如前所述，在进程中可能无法取消查询。CursorLoader 会尽快前来。它尝试取消查询，并在取消失败时尽可能快地丢弃结果。

它还会调用注册的回调处理函数，并通知它加载器已经取消。这是第 3 个加载器生命周期事件，是通过将清单 4.20 中视图适配器的游标替换为 null 来处理的。

加载器演示了几种巧妙使用 AsyncTask 的方式。

- 对客户端隐藏了任务细节。虽然实现这一点稍微有点复杂，但最终代码与我们最初的想法很接近。当查询完成时，会有一个包含查询和方法调用的对象。让查询在后台运行的机制几乎是不可见的。
- 任务应尽可能是可取消的。当请求的游标不再有用时，就要尽可能快地将它丢弃。它不会为不需要的数据保留内存，也不会让 CPU 为它忙碌。
- 加载管理器巧妙地使用了 AsyncTask 的生命周期。如果某个数据集的加载器可能对未来的 Activity 有用，管理器就会让它运行到完成，即使请求数据的

Activity 已经消失很久了。相反，当它可用时，管理器就会将加载的数据传递给 Activity 的新实例。

4.4　AsyncTask：哪里出错了？

开发人员如何错误地使用了 AsyncTask？其实这很容易理解。当有抱负的 Java 开发人员在 IDE 中创建了第一个 Java 应用程序时，他可能会看到 IDE 提供的框架代码，如清单 4.21 所示。

清单 4.21　一个 Java 应用程序框架

```
public class AwesomeNewApp {

    public static void main(String… args) {
        // …
    }
}
```

如果这名程序员决定进行 Android 开发，那么这个 IDE 会为他的第一次 Android 应用程序尝试产生如清单 4.22 所示的代码。

清单 4.22　一个 Android 应用程序框架

```
public class AwesomeNewApp extends Activity {

    protected void onCreate(Bundle state) {
        // …
    }
}
```

“啊哈！”他说，然后脑海中浮现出表 4.1 所示的模型。

表 4.1　一个简单的模型

角　色	桌面应用程序	Android 应用程序
应用程序文件	.jar	.apk
应用程序类	myClass	myActivity
初始化方法	public static void main	protected void onCreate

在这个模型基础上，开发者将越来越多的功能塞入 Activity 中。当然，有经验的

开发人员会将一些行为放入新的包和类中来保持模块化。然而，Android 新手程序员可能会创建将一个或者多个 Activity 作为设计中心的应用程序。

要创建好的 Android 应用程序，理解它们是 Web 应用程序（正如第 3 章所指出的）会大有帮助。考虑到表 4.2，开发人员会创建更稳定、行为更一致、更令人愉快的程序。

表 4.2 Android 模型

角　色	Web 应用程序	Android 应用程序
应用程序文件	.war	.apk
视图/控制器	servlet	activity
业务层	管理器/会话 bean	服务
持久层	DAL/模型	提供者

在更精确的模型中，AsyncTask 很有意义：它们是快速处理小而慢任务的方式，这样 UI 就可以非常快速地反应。

模型的更新与远程数据进行同步和业务计算都不属于本章的讨论范围。它们属于第 6 章描述的 Service。

4.5 小结

本章深入探讨了 AsyncTask 的体系结构。AsyncTask 是一种将小段代码的执行从主线程移动到工作者线程上的工具。本章探讨的要点如下。

- AsyncTask 是类型安全的模板，必须被子类使用。为了执行任务，AsyncTask 要创建子类的一个新实例并调用其 `execute` 方法。一个实例只能执行一次。
- AsyncTask 的模板方法 `doInBackground` 在执行器的工作者线程上运行。默认的执行器 `SERIAL_EXECUTOR` 每次按顺序只执行一个任务，但不一定是在一个线程上执行的。它可能会使用其他执行器。
- 将代码封装在 AsyncTask 上，然后在后台线程上执行它，这是不可能的。由于被移动到 AsyncTask 上的代码可在另一个线程上执行，因此很多普通的 Java 关键字（`break`、`continue`、`throw` 和 `return`）会有完全不同的含义。然而更重要的是，Java 语言的基本块结构实际上保证了这些任务会包含并发错误。

版权所有，严禁传播，违者自负法律责任！

- AsyncTask 的生命周期和与它们交互的 Android 组件完全不同。这两种生命周期的不一致可以导致内存泄漏和应用程序失败。
- AsyncTask 在符合其初衷的地方非常高效：小的自发性任务，或者可取消的任务。Android 加载器 `CursorLoader` 是有效利用 AsyncTask 的示例。

AsyncTask 过多地在不符合初衷或者非常不适合的地方使用。这种滥用多数源于开发者对 Android 体系结构的错误理解。将 Android 作为 Web 应用程序看待，你就能意识到 AsyncTask 适用于那些小的局部任务，而不是大型任务。

版权所有，严禁传播，违者自负法律责任！

第 5 章 Looper/Handler

一群蚂蚁排着队行进！嗨！嗨！

儿歌

虽然 AsyncTask 是大多数开发人员开始使用 Android 时遇到的第一个并发结构，但它们并不是其核心并发结构。在 Android 中，基本的并发结构是 Looper/ Handler。

5.1 Looper/Handler 简介

和 Android 内部架构的其他重要部分一样，Looper/Handler 的灵感可能来自 BeOS 操作系统——尤其是，其 `Boolper` 和 `BHandler` 类。

Looper 框架是第 2 章中安全发布习惯用法的扩展，如图 5.1 所示。这个习惯用语由一个下拉框和下拉框上的锁组成。

这个习惯用法的重点是，通过一个正确同步的中间件，将一个本身不是线程安全的对象安全地从一个线程发布到另一个线程上。只要发送者不保留所传递对象的引用，传输就是安全的，当然，所传递的对象不会保留对源线程所拥有对象的引用。

为了简化，第 2 章中的示例在传输过程中使用一个单项下拉框来保存传递的对象。Android Looper 是安全发布用法的扩展，它使用排序队列（而不是简单的下拉框）发布对象。

Looper 框架如图 5.2 所示。工作者线程是将要发布对象的线程。MessageQueue 取代下拉框。客户端捕获 MessageQueue 的锁，并将要传递的对象加入队列中。工作者线程捕获同一个锁，并将对象从队列中删除。与第 2 章习惯用法的唯一区别是 MessageQueue 拥有很多项，并且会建立被发布的顺序。

版权所有，严禁传播，违者自负法律责任！

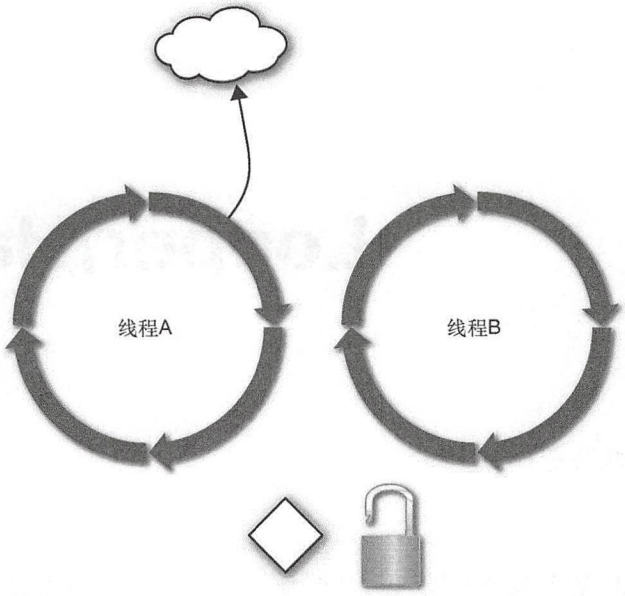


图 5.1　安全发布

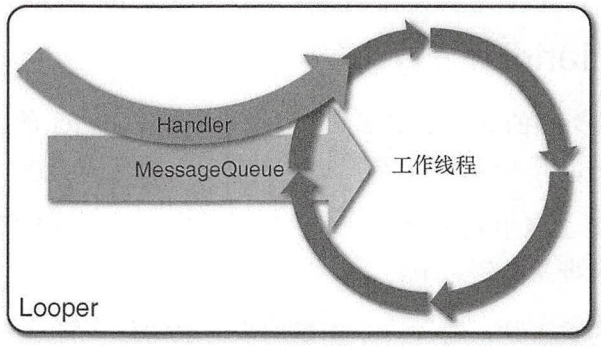


图 5.2　Looper 框架

Handler 是 Looper/Handler 框架的附加功能。它将底层线程和 MessageQueue 的一些细节抽象化。

Looper 框架由 4 个 Java 类和一些 C 代码组成。Java 类如下所示：

- `android.os.Looper`
- `android.os.Handler`
- `android.os.MessageQueue`

版权所有，严禁传播，违者自负法律责任！

- android.os.Message

还有一个辅助类：

- android.os.HandlerThread

5.1.1 Looper/Handler 的基础

Looper 并不只是一个简单的发布机制，它的目标是将少量工作和任务从源线程转移到 Looper 的工作者线程中执行。它是一个由 Java 工作者线程驱动的事件队列。使用 Looper 初始化时，线程会获得两个特性。

- 对 MessageQueue 实例的引用。
- 从工作者线程的 run 方法中调用的队列管理器 Looper 的 loop 方法。

当 Java 线程初始化为 Looper 时，它的 run 方法会调用 Looper.loop。loop 方法是一个无限循环，它可将任务从 Looper 的 MessageQueue 中删除，并从 Looper 的工作者线程中回调将其入队的 Handler 来处理它们。

走查一个任务的执行会说明这个过程。假设某个线程（在这里称为源线程）有一个可能已经被工作者线程执行并用 Looper 初始化的任务。图 5.3 演示了这一点。

源线程（见图 5.3 中的窗格 1）先为目标 Looper 创建一个新 Handler：

```
Handler handler = new Handler(looper);
```

变量 looper 是对工作者线程 Looper 的引用。赋值后，变量 handler 会包含对 Handler 对象的引用，反过来 Handler 对象将持有对工作者线程 MessageQueue 的引用。

一个 Handler 可以用于将很多任务加入队列中。当源线程中运行的代码有对 Handler 的引用时，它就可以使用 Handler 创建尽可能多的任务，并将这些任务加入队列中进行处理。

一旦源代码有了对目标工作者线程 Handler 的引用，它就使用该 Handler 从池中获取 Message 对象。它会将想要在工作者线程中执行的任务连接到 Message，然后使用 Handler 将 Looper 的 Message 加入队列中（见图 5.3 中的窗格 2）。

```
Message msg = handler.obtainMessage(DELIVER_STUFF, stuff);
handler.sendMessage(msg);
```

版权所有，严禁传播，违者自负法律责任！

Handler 使用正确的同步将对任务引用的 Message 加入队列中，它是在 Looper 的 MessageQueue 上的（见图 5.3 中的窗格 3）。当然，前几章的规则都适用。如果传递的任务对象本身不是线程安全的，那么源线程就必须删除对其的所有引用以及它引用的所有内容。同样，任务不能持有对源线程所使用对象的任何反向引用。

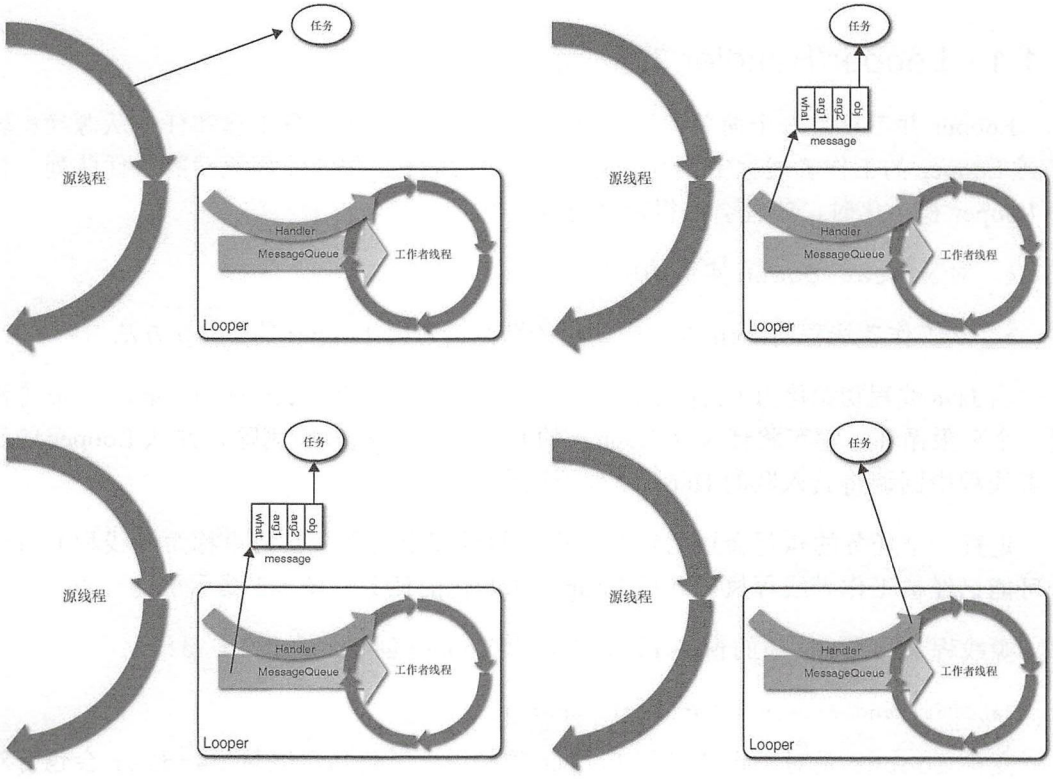


图 5.3　使用 Handler 将消息传送给 Looper

到目前为止，所有代码都是在源线程上运行的。然而，如果 Message 加入到 Looper 的 MessageQueue 中，它的所有权就会转移给 Looper 的工作者线程。最后，在 Looper 的 loop 方法中，将 Looper 工作者线程中的 Message 从 MessageQueue 中删除。Looper 将从队列中删除的消息返回给将它添加到队列中的 Handler（仍然在工作者线程中）进行处理（见图 5.3 中的窗格 4）。Handler 负责在工作者线程的回调方法中处理 Message 和其相关任务。

5.1.2　委托执行

Looper/Handler 框架为任务处理提供了两个习惯用法。虽然有一个底层机制支持

版权所有，严禁传播，违者自负法律责任！

这两种习惯用法，但它们是完全不同的。在尝试了解框架时，由于环境限制，将它们看成完全不同的工具是很有帮助的，它们可塞入一个实现中。

1. 发布 Runnable

使用 Looper/Handler 将执行委托给另一个线程的最简单方法是使用 Handler 中以 post 开头的方法集。这些方法将参数作为 Runnable，它会成为传输给工作者线程的负载。当其中的一个 Handler post…方法被调用时，Handler 内部就会获取一个 Message 对象并连接到 Runnable，然后将 Message 添加到 Handler 所连接的 Looper 的 MessageQueue 中。

如图 5.3 所示，当使用目标 Looper 从 MessageQueue 删除一条 Message 时，它会传递回最初将它添加到队列中的 Handler 进行处理。

Handler 从内部识别 Message 连接的 Runnable，然后提取它，并调用其 run 方法。清单 5.1 演示了在短暂的延迟后开始一个动画的 Runnable。

清单 5.1 简单的延迟动画

```
private uiHandler;

@Override
@SuppressLint("HandlerLeak")
public void onCreate(Bundle state) {
    super.onCreate(state);

    // ...

    uiHandler = new Handler();
}

// ...

private void revealButtonDelayed() {
    uiHandler.postDelayed(
        new Runnable() {
            @Override
            public void run() {
                button.setVisibility(View.VISIBLE);
```

版权所有，严禁传播，违者自负法律责任！


```
                ViewAnimationUtils
                    .createCircularReveal(button, ctr, ctr, 0, buttonDiameter)
                    .start();
            } },
        BUTTON_DELAY);
}
```

Handler 是在没有将显式目标 Looper 指定为 Handler 构造函数（在默认情况下，线程使用运行构造函数的 Looper）的情况下创建的。如果当前线程没有初始化为 Looper，在没有指定目标 Looper 的情况下尝试创建一个 Handler 会导致运行时异常。

在清单 5.1 中，变量 uiHandler 引用的 Handler 指向与主线程关联的 Looper。这是有保证的，因为 onCreate 方法总是在主线程上运行的。它有效的原因是 Android 的主线程是 Looper 的工作者线程。静态方法 Looper.getMainLooper 是为主线程获取同一个 Looper 的另一种方式，它几乎可以用于所有环境中。

清单 5.1 中的代码提交了一个 Runnable，它会在 BUTTON_DELAY 后启动一个显示 button 的动画。清单 5.1 中的代码存在一个严重缺陷，读者对此应该已经很熟悉了。试考虑如果在用户旋转设备前执行这段代码会发生什么。

适用于 AsyncTasks 的所有利害关系在这里同样适用。失败的直接原因是当它尝试启动按钮时，动画框架抛出的异常，其中按钮不再是活动视图层一部分。要记住的是，当设备旋转时，整个视图层都会丢弃，取而代之的是一个新的视图层。

此外，回想一下，由于清单 5.1 中的 Runnable 是匿名类，因此它拥有对其周围 Activity 的引用。在 Runnable 释放前，Activity 不能被垃圾回收，即需要等待 BUTTON_ DELAY 毫秒。

这些问题的解决方式有多种。需要对细节有清楚的了解，才能做出正确的选择。人们可能会天真地使用广泛而通用的并发模式。然而，在特殊情况下，有更轻量级的解决方案，如清单 5.2 所示。

清单 5.2　改进的延迟动画

```
Runnable animator;
private uiHandler;

@Override
```

版权所有，严禁传播，违者自负法律责任！


```
@SuppressLint("HandlerLeak")
public void onCreate(Bundle state) {
    super.onCreate(state);

    // ...

    handler = new Handler();
}

@Override
public void onPause() {
    if (null != animator) {
        handler.removeCallbacks(animator);
        animator = null;
    }
    super.onPause();
}

// ...

private void showButton() {
    if (animator != null) { return; }
    Runnable anim = new Runnable() {
        @Override
        public void run() {
            button.setVisibility(View.VISIBLE);
            ViewAnimationUtils
                .createCircularReveal(button, ctr, ctr, 0, buttonDiameter)
                .start();
            animator = null;
        }
    };

    uiHandler.postDelayed(anim, BUTTON_HIDE_DELAY);
    animator = anim;
}
```

清单 5.2 中的所有代码都在**单个**线程（主线程）上运行。不需要复杂的同步或者

版权所有，严禁传播，违者自负法律责任！

锁。此外，由于一个线程执行所有的代码，因此每个方法都是原子的。执行动画 Runnable run 方法的线程必须从该方法返回后，才能执行其他方法。特别是，这意味着 run 和 onPause 方法的执行是互斥的。

考虑清单 5.2 所示的可能状态。在初始状态中，变量 animator 为空，并且调用 onPause 会调用 super.onPause。在初始状态中调用 showButton 将执行整个方法，将延迟动画添加到队列中，并将 animator 设置为非空。将 animator 设置为非空的唯一方法是当动画排队时对其进行设置。

有两种方法可以返回初始状态。第一种方法是通过 animator 的正常完成。主线程将包含动画 Runnable 的消息从队列中删除，并执行它来启动动画。一旦这么做了，无动画的任务仍在排队，并且 animator 会重置初始状态为空。图 5.4 即为这两种方法的典型执行。

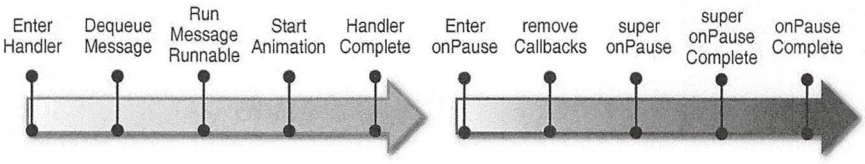


图 5.4　典型的执行

返回初始状态的第二种方法是，当动画任务在消息队列中时调用 onPause。在这种情况下，animator 为非空，因此 onPause 会从消息队列中删除它引用的消息，并将 animator 设置为空——初始状态。将 animator 设置为空的唯一方式是在将消息从队列中删除时。总而言之，只有当动画 Runnable 在队列中时，animator 才是非空的。

还要注意，当在 onPause 中将消息从队列中删除时，Runnable 就没有其他悬空引用了。因此，Runnable 可以被垃圾回收，并且不会妨碍有隐含指针的 Activity 被垃圾回收。

清单 5.2 中的所有代码都必须在主线程上运行，因为它处理的是视图对象。绝对不能从其他线程上触摸 Android 视图对象。由于它们在同一个线程上运行，因此这两个方法（Runnable 的 run 和 Activity 的 onPause）是异步运行的，但它们不是并发的。

考虑一下，如果 run 和 onPause 并发运行，解决方案会有什么不同。至少，变量 animator 应该是 volatile 的，因为这样可以从多个线程中访问它。然而更麻烦的

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

是，如果 Activity 的 onPause 和动画 Runnable 的 run 方法可以在不同线程上运行，那么它们彼此之间就不是原子的。它们之间就不会有“先于发生”关系。

这种情况下，在将 Runnable 从 MessageQueue 中删除和其 run 方法完成期间，可能会存在竞争。假设 onPause 方法会在没有额外同步的情况下取消所有动画。如果它正好在此期间执行，那么就要保证 run 方法不会在完成后调度一个新动画。图 5.5 说明了这种竞争条件：工作者线程从队列中提取出一条消息，并在 onPause 完成后开始一个动画。

这种假设情况（onPause 和 run 在不同线程上运行）是更难一点的。但它绝对不是不可逾越的。几行代码就可以修复这个问题。然而，重点是由于 onPause 和 run 必须在同一个线程上运行，因此最好的一种代码是：你不需要的代码。在清单 5.2 中使用 Looper/Handler 并不意味着有大量同步和并发结构。相反，在对实际情况进行清晰的评估后会发现，一种非常简单的解决方案也是很好的。

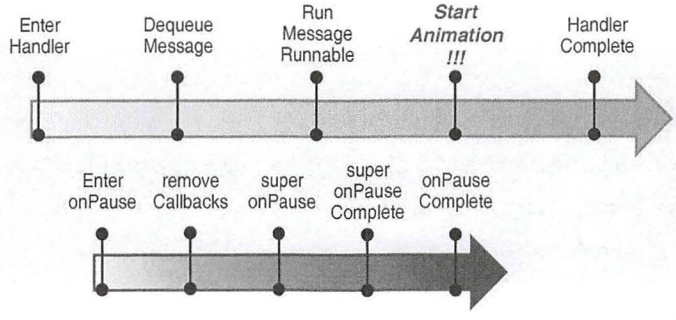


图 5.5 竞争条件

2. 将一条消息入队

虽然发布 Runnable 非常方便，但重要的问题是以它们作为委托执行的方式使用。

第一个问题是架构上的。任何组件都可以将 Runnable 发布到任何 Handler 上的应用程序，这可能很快就变成一个噩梦。在其他地方或其他时间执行的代码可能很难调试。

此外，由于它是 Android 中核心的并发机制，因此几乎所有影响应用程序的事件都会表示为由 Looper 处理且添加到队列中的消息。所添加的来自网络的所有事件，例如键盘、地理定位传感器和其他资源，都可能会出现。例如，在一些动画中，每隔 17ms 左右就会发生一些事件。总计每秒会发生几百次事件。

非卖品，仅供非商业用途或交流学习使用

那是很多事件。如果每个事件都需要创建一个新 Runnable 对象，然后在不需要这个对象时作为垃圾回收它，垃圾回收器将会非常忙。一种很明智的优化是——Looper/Handler 框架支持一个可回收的消息对象池。消息对象本身可以识别要执行的任务并提供一些参数，而不是传输 Runnable。

有多种从池中获取消息的方式。最方便的是从将它入队并从处理它的 Handler 中获取它。Handle 上名为 obtainMessage 的重载方法集从池中检索并初始化消息。例如，

```
Handler handler = new Handler(looper);
handler.obtainMessage(OP_SHOW_BUTTON).sendToTarget();
```

对 obtainMessage 的调用返回一条 Message，其 what 字段初始化为 OP_SHOW_BUTTON（Java int），其 target 字段（包含将消息传递进去进行处理的 Handler 所引用的字段）是用对 handler 引用的初始化。调用 sendToTarget 会导致消息交换给 handler，在 Looper 的消息队列中进行入队操作。

注意

重载方法是有多个签名但只有一个名字的方法。例如，Handler 方法 obtainMessage()、obtainMessage(int)、obtainMessage(int, int, int)、obtainMessage(int, int, int, Object)和 obtainMessage(int, Object)都是重载方法 obtainMessage 的示例。

使用这个习惯用法时，与消息相关的行为会在处理消息的 Handler 中，而不是在这条消息中（就像在 Runnable 习惯用法中一样）。清单 5.3 是对只使用消息实现的清单 5.2 所实现特性的实现。这个习惯用法和其 switch 语句在 Android 代码中极为常见。

清单 5.3　消息延迟动画

```
private static final int OP_START_ANIMATION = -1;

private boolean animating;
private Handler handler;

@Override
@SuppressLint("HandlerLeak")
public void onCreate(Bundle state) {
```

非卖品，仅供非商业用途或交流学习使用

```
    super.onCreate(state);

    // ...

    handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case OP_START_ANIMATION:
                    animateButton();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    };
}

@Override
public void onPause() {
    if (animating) {
        handler.removeMessages(OP_START_ANIMATION);
        animating = false;
    }
    super.onPause();
}

// ...

private void showButton() {
    if (animating) { return; }
    handler.sendMessageDelayed(
        handler.obtainMessage(OP_START_ANIMATION),
        BUTTON_HIDE_DELAY);
    animating = true;
}
```

非卖品，仅供非商业用途或交流学习使用

```
void animateButton() {
    button.setVisibility(View.VISIBLE);
    ViewAnimationUtils
        .createCircularReveal(button, ctr, ctr, 0, buttonDiameter)
        .start();
    animating = false;
}
```

该实现的分析与清单 5.2 几乎相同。唯一的变化是 showButton 现在只将其 what 字段中包含操作码的可回收消息入队，而不是传递给一个执行的对象。Handler 中的 case 语句启用操作码的值来调用合适的方法：在 Runnable 习惯用法中，已经成为 Runnable 的 run 方法内容的代码。handleMessage 完成后，Handler 会自动返回消息到消息地。

虽然可回收消息的习惯用法是轻量级的，但它受到的限制很多。Runnable 几乎有可以想象得到的所有行为，和闭包一样，它可以继承周围环境中的状态。消息只是数据，它只能调用已经存在于 Handler case 语句中的行为，并且只能传输专门连接的数据。然而，这些限制可以让代码更清晰，线程更安全。虽然选择权属于开发人员，但除了最简单的情况以外，这种轻量级的习惯用法更可取。

必须要重申的是，这 3 个清单中使用的 Handler 都可能会导致内存泄漏。由于它有一个内部类，因此它有一个指向包含 Activity 的隐式指针。使用 Handler 入队的消息持有指向该 Handler 的指针。消息入队后，MessageQueue 就会持有对这条消息的引用。MessageQueue 是长时间存在的对象，被启动 Looper 的线程所引用。在将给定 Handler 的最后一条消息从队列中删除前，Handler 和 Activity 都不符合垃圾回收的条件。

这对本章的示例并不重要，因为在 Activity 有资格被破坏前，onPause 就会从队列中删除指向 Handler 的所有消息。然而，如果修改代码的下一个开发人员不那么小心，就很容易泄漏 Activity。为了防止这种可能性，Android Lint 推荐所有 Handler 都是静态内部类，并且最多只有其 Activity 的 WeakRefererce。

这是 Looper/Handler 框架的核心：单个线程为一个队列服务，在这个队列中可以使用 Handler 将消息加入队列中。Handler 会对如何处理消息进行非常细粒度的控制，它既是最终处理的生产者（这个生产者会在地线程上调用消息添加到队列中），又是在 Looper 的工作线程上运行并处理这些消息的消费者。

非卖品，仅供非商业用途或交流学习使用

5.2 一些细节

前面的几节提供了使用 Looper/Handler 框架所需的大部分信息。然而，有一些细节可以帮助你更有效使用它。

5.2.1 Handler 和消息

`Messoge` 是一个小而简单的对象：基本上是一个 POJO（简单的 Java 对象）。它的大多数字段实际上都是由 Handler 管理的。

消息本身不是线程安全型的。这些消息的字段都不是 final 或 volatile 的。一个入队线程不能持有加入其他线程队列的消息引用。

消息的示意图如图 5.6 所示。

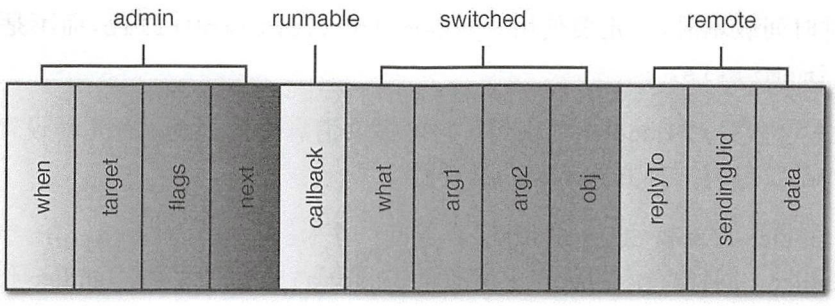


图 5.6 消息

如图 5.6 所示，有 4 组具有相关用途的字段。

- admin group：由框架使用用于内部管理。
- runnable：可运行的执行。
- switched：可切换的执行。
- remote：进程间通信。

1. admin 字段

框架使用管理字段进行基本消息处理。所有字段都是框架私有的（包保护的）。最感兴趣的是何时处理消息的 `when` 和前面提到的 `target`，它是对提交消息用于处理

非卖品，仅供非商业用途或交流学习使用

Handler 的引用。

when 字段实现了定时队列。正如前面的清单所示，消息可以在未来的某个时间加入队列中执行。Looper/Handler 消息队列是由 when 字段排序的，只有当 SystemClock.uptimeMillis 测量的当前时间大于等于已在消息 when 字段中排序的时间时，Looper 才会进行出队操作并处理消息。一些重载的 Handler 方法支持定时延迟。

- postAtTime(runnable, ..., uptimeMillis)
- postDelayed(runnable, delayMillis)
- sendMessageAtTime(message, uptimeMillis)
- sendMessageDelay(message, delayMillis)

虽然简单的 post...和 send...方法只能在当前时间设置，但这些方法可以让框架的客户端在特定时间或者当前时间稍后的某个时间提交要执行的消息。当使用这些 post 方法的绝对时间版本时，一定要使用 SystemClock.uptimeMillis，而不是 System.currentTimeMillis。

target 字段是对 Handler 的引用，它可提供消息进行处理。有几种设置目标的方式。有趣的是，只有一种方式是无关紧要的。

调用 Handler 的重载 obtain...方法会返回一个 Message 它的 target 字段指向被调用的 Handler。同样，Message 中所有静态 obtain...方法都将 Handler 作为第一个参数。实际上，Handler 只是调用 Message 方法，并将 this 作为它们的第一个参数来传递。甚至可能有第 3 种可能性，Message 实例中的 setTarget 方法。

这些设置消息目标的方式都没有任何意义。当消息加入队列中进行处理时，Handler 会将 target 入队并作为进程的一部分来指向自己。将消息添加到队列中的 Handler 总是其目标。清单 5.4 是一个会混淆的代码清单，但它会通过将 what 设置为 OP_CONFUSION 来将消息添加到 looper1 的 MessageQueue 中，以便 handler1 进行处理。

清单 5.4　目标设置

```
Handler handler1 = new Handler(looper1);
Handler handler2 = new Handler(looper2);

handler1.sendMessage(handler2.obtainMessage(OP_CONFUSION));
```

非卖品，仅供非商业用途或交流学习使用

虽然消息是从 handler2 中获取的，但使用 handler1 将它入队会将 target 设置为 handler1。

注意

Message.setTarget()?你是认真的吗?

有人想知道公开该方法有什么用。滥用该方法没有任何好处。

考虑:

```
Message msg = Message.obtain();
handler.sendMessage();
msg.setTarget(null); //!!!
```

最后一个管理字段 flags 主要用于 FLAG_IN_USE 位。这个位用于追踪 Message 的生命周期。它在消息释放到消息池中时进行设置，然后从池中删除时清除，并在它添加到队列或者释放到池中时检查。如果它在检查中被设置，就会抛出一个异常。这保证消息不会双重释放到池中，一旦它们释放就不会再次添加到队列中了。只有当消息在池中并且使用时，这个位才会设置。另一个标志 FLAG_ASYNCHRONOUS 影响调度，稍后我们会讨论它。

2. 消息字段

这里分为“runnable”和“switched”的字段组表示 Message 的两种手段，这在清单 5.2 和清单 5.3 中讨论了。如果 callback 是非空的，那么 Handler 就会完全忽略 switched 字段，并为 callback 指向 Runnable 调用 run 方法。这种行为是可以改变的。它是 Handler 方法 dispatchMessage 中实现的——不是最终的，可以重写。

如果 callback 为空，dispatchMessage 就会将消息解释为 switched。交换的消息以一种方式进行处理。如果 Handler 实例是用 Handler.Callback 的一个实例初始化的，那么回调的 handleMessage 方法就会调用来传递消息。如果 Handler 实例没有 Handler.Callback，或者调用 Handler.Callback. handleMessage 返回 false，就会使消息调用 Handler 自己的 handleMessage 方法。基类 Handler 实现了 handleMessage，但其实现是空的：它只是忽略了所有消息。没有传递 Handler.Callback 或者重写 handleMessages 的 Handler 子类可用于发布 Runnable。

交换的消息通常由 Message what 字段的 switch 来处理。Android 源本身(ASOP)使用的就是 Looper/Handler 框架。

非卖品，仅供非商业用途或交流学习使用

arg1 和 arg2 字段都是简单的 int 值，可作为 what 字段选定执行代码的参数来使用。第 3 个参数 obj 是对 Object 的引用，因此可以使能任意多个附加参数。它还意味着所有常见线程安全的约束。

3. 剩余字段

消息中剩余的字段用于进程间的通信。我们将在第 6 章中讨论它们。

5.2.2 启动 Looper

正如本章开头所述，Looper 是用 Java 线程和 MessageQueue 初始化的队列管理器。不管一个 Looper 连接着多少 Handler，只有一个 Looper 与给定的线程相连。和 Java 的 Executor 框架不同，Looper 框架只支持为一个队列服务的线程。

将线程初始化为 Looper 的步骤如下。

1）创建并启动线程。

2）从运行线程上调用静态方法 Looper.prepare。

3）执行所有额外的初始化。

4）从运行线程上调用静态方法 Looper.loop。该方法在 Looper 停止前不会返回。

清单 5.5 演示了一个简单的 Looper。

清单 5.5　Looper 的创建

```
Thread looperThread = new Thread() {
    @Override
    public void run() {
        Looper.prepare();
        Looper.loop();
    }
};
looperThread.start();
```

虽然现在 looperThread 作为 Looper 运行，但它并不是很有用。Looper/Handler 框架不提供查找与任何线程相关联的 Looper（如果存在）的方法。由于没有查找与

非卖品，仅供非商业用途或交流学习使用

looperThread 相关的 Looper 的方法（即使有），因此没有将它与 Handler 连接的方法，因此也就不存在使用它提交或者处理工作的方式了。

Android 文档建议创建 Handler 作为初始化 Looper 线程的一部分，如清单 5.6 所示。

清单 5.6 Looper 的创建

```
public volatile Handler mainHandler;

// ...

    Thread looperThread = new Thread() {
        @Override
        public void run() {
            Looper.prepare();
            mainHandler = new Handler();
            Looper.loop();
        }
    };
    looperThread.start();
```

清单 5.6 中存在习惯用法的一种变体。在这种变体中，通过使用 Looper.myLooper 方法，用引用 Looper 的发布代替将 Handler 引用放入变量 mainHandler 中的发布。myLooper 方法返回初始化当前线程的 Looper，它可以在调用 Looper.prepare 后的任何时间使用。

虽然这两种变体几乎相同，但当 Looper 有特殊的目的时，清单 5.6 中的代码可能更清晰。在进入循环前，初始化的标准 Handler 可以显式地实现这个目标，从而简单而清楚地表示代码的功能。

发布一个 Handler 的引用就足够了，即使稍后还需要添加其他 Handler。调用 Handler 方法的 getLooper 会返回对 Handler Looper 的引用，可用于初始化其他 Handler。

清单 5.6 中的代码包含重要的数据竞争。正如所介绍的那样，没有简单的方法可以说明对 mainHandler 的赋值何时发生。Thread 的 start 方法调度新线程，并立即返回。它不能保证新调度的线程何时开始执行。对 mainHandler 的赋值会在 start 返回时立即看到，或者只有几百条语句执行完后才能看到。

非卖品，仅供非商业用途或交流学习使用

Android 的类 HandlerThread 会避免竞争，并让创建新 Looper 成为一个非常简单的过程，如清单 5.7 所示。

清单 5.7　Looper 的创建

```
HandlerThread looperThread = new HandlerThread("AlternateLooper");
looperThread.start();
Looper looper = looperThread.getLooper(); // may block!
```

HandlerThread 从内部看很像清单 5.6。它在 getLooper 方法中添加一个额外的同步机制，该方法会一直阻塞直到 HandlerThread 开始执行并初始化其 Looper 为止。该方法返回时，保证 looper 是非空的。显然，由于阻塞，它应该从非时间敏感的代码中调用 HandlerThread.getLooper——这也许是在组件初始化时。

注意

不要忘记启动线程！

清单 5.7 中的代码会永久阻塞，除非调用 looperThread 的 start 方法！简单地创建线程并不会调度它执行。

5.2.3　Native Looper

每个 Looper 都有一个看起来像 Java 框架的镜像，但其实它是用 C/C++编写的 native 影像。这个影像有对每个 Looper/Handler 框架 Java 组件的模拟：Message、MessageQueue、Handler 和 Looper。令人惊讶的是，调度 native Looper 的线程和 Java Looper 相同。因此，Looper 的工作者线程实际上为两个队列服务：Java 队列和 native 队列。Native 队列在 Java 队列之前轮询。

当没有消息准备执行时，native Looper 就会交出 CPU。当任何一个队列中都没有可调度的消息时，native Looper 就会产生使用 Linux **epoll** 机制的处理器。epoll 是一个内核原语，它使能线程的等待在任何文件描述符集中新数据的出现。由于它用这种方式来产生处理器，因此可以通过另一个线程异步唤醒 Looper，这个线程将文件写入让 Looper 在其中等待的文件描述符中。

epoll 机制也支持定时等待。该机制允许线程一直等待，直到新数据到达或者经过指定的时间，以先发生者为准。Looper 使用 epoll 定时等待机制“睡觉”，调度自己在 MessageQueue 中的下一条消息被调度处理时重启。

这是安全的，即使唤醒时间距离未来还很远。如果消息是其他线程加入队列中的，并且必须在睡着的 Looper 线程被调度唤醒之前调度，那么将一个字节写入标准管道（不用惊讶，名为 mWakeWritePipeFd）就会将它唤醒。

将任务添加到 Looper native 队列中的唯一方法是使用 native 框架。因此，除了与主线程有明显区别的 Looper 外，大多数 Looper 在它们的 native 队列中没有做任何有意义的工作。轮询 native 队列只是次要开销。

从 Android 6.0（Marshmallow）起，Looper/Handler 框架引入了 Java 中的几种新方法。其中有一种为预定任务查询队列的方法 MessageQueue.isIdle，它可以用于直接确定 Looper 是否空闲。

还有两种直接 Java 访问 epoll 机制的新方法：MessageQueue.addOnFile-DescriptorEventListener 和 MessageQueue.removeOnFile DescriptorEvent-Listener。这两种方法都使能 Java 代码在 native Looper 的 epoll 文件描述符等待集合中注册新的文件描述符和回调。

5.2.4 调度和同步屏障

如果不讨论任务调度的细节，那么对细节的讨论就是不完整的。任务调度是在 MessageQueue 方法 next 中实现的。

也许首先注意的是，由于 MessageQueue 的队列是一种由执行时间排序的简单链表，因此将任务入队的计算复杂度为 O(n)。由于 n 通常很小，因此大多数情况下开销是可以忽略的。然而，在不指定运行时间而调度任务的正常情况下，几乎肯定会引起整个队列的浏览。在一个大量使用的 Looper 上，如与主线程相关的 Looper，这种开销就很明显。

如前所述，Looper 通过调用本地代码生成处理器。在 Java 框架端的队列轮询代码将控制权传递给准备产生 CPU 的本地代码前，它会调用任何一个注册的 MessageQueue. IdleHandler。这会提供一种机制，代码通过该机制在处理任务时可以释放所需要的资源。例如，网络接口可以将网络连接打开作为一部分，它处理从队列中传递的第一个请求。然后，当最后一个请求完成后，它可以注册 MessageQueue. IdleHandler 以关闭连接。

然而，任务调度最有趣的转折点是**同步屏障**。同步屏障是在 JellyBean 发布时出现在 Android 中的。同步屏障用于停止 Looper，这样并发任务就可以同步执行了。尤其

是，Android 5.0、Lollypop、API level 21 和材质设计引入了一种新的 UI 机制，它用于优化动画和高性能的图形。这种架构大量使用了一种新的与主线程分离的专用渲染线程。框架需要一种方式以暂停 UI 相关主线程任务的执行，而渲染线程则通过遍历视图树来渲染视图。

同步屏障为任务引入了一种新维度。除了 post.../send...的区别（Message/Runnable 处理）外，从 JellyBean 开始消息也是同步或者异步的。异步消息是不可以同步屏障处理的消息。Android API level 22 提供了让开发人员可以控制任务属性的方法 `Message.setAsynchronous`。

同步屏障作为具有空目标的消息来实现。无法用可见接口正确地创建这种消息。只能使用方法 `MessageQueue.postSyncBarrier` 来创建。从 Android 6.0 版本开始，该方法就用`@hide` 隐藏起来了，只能从遍历视图层的代码中调用。同步/异步的区别只与主线程有关。

当 MessageQueue 任务调度代码遇到目标较少的消息时，它会继续浏览任务队列，但会忽略还没有标记为异步（它们的标志设置为 `FLAG_ASYNCHRONOUS`）的消息。如果有准备好可以执行的异步消息，它们会正常调度。如果没有任何异步消息准备执行，那么 Looper 就会产生处理器，并为队列中第一个异步任务的执行时间调度唤醒。如果队列中没有异步消息，那么睡眠时间就是−1。

解除停止阻塞 MessageQueue 的唯一方法是显式地删除同步屏障。隐藏的 MessageQueue 方法 `removeSyncBarrier` 会删除屏障，唤醒线程，并将唤醒时间精确到下一个可调度任务的时间。

使用同步屏障会对在主线程上调度的准确性产生惊人的影响。表 5.1 显示了 3 种对于不同类型任务执行时间的标准差。测量的测试代码未来会有 10ms 的时间用于消息传递，并在执行时计算预计时间和执行时间的差别。表中显示的测试是在运行一个简单的全屏动画时执行的，它们在几个不同的设备上运行。虽然设备之间的标准差不同，但表中显示的是典型数字。

表 5.1　执行时间标准差（典型设备）

消息类型	标准差/ms
主线程，同步	大约为 2.00
主线程，异步	大约为 0.20
工作者	大约为 0.17

在新创建的 Looper 上只运行测试任务，其标准差小于 0.2ms。在主线程上使用标记为异步消息的测试会产生一个比较高的标准差。这是可预计的，因为运行动画的主线程已经很忙了。

当任务在主线程上运行且没有标记为异步时，会产生很大的影响。在这种情况下，标准差会增加一个数量级，成为 2ms。

5.3 小结

Looper 是 Android 中基本的并发结构。它在安全且轻量级的框架中支持可调度的异步任务。

Looper/Handler 框架非常复杂。它有两部分，即 Java 和 Native。Native 部分以 Linux `epoll` 机制为基础，并支持等待指定的时间，或者新数据被写入其中一个文件描述符集合中。

框架支持几种消息的执行。它支持直接执行 Java Runnable，也支持一种更节俭的模式，在这种模式中要执行的代码（通常在 Handler 方法 `handleMessage` 的 `case` 语句中）是通过使用操作码在消息的 `what` 字段中选择的。

Android 框架初始化主线程，该线程是作为使用 Looper 为新应用程序启动新进程的一部分而创建的。这个主线程就是标准的 Looper 工作者线程，即 UI 组件（几乎包括所有内容，除非有特殊安排）运行的线程。主线程的 Looper 可以通过 Looper 方法 `getMainLooper` 获取。

细心的读者会在考虑完 Looper/Handler 框架后，从 Looper/Handler 方面重新实现 AsyncTask。实际上，有证据表明这就是最初实现 AsyncTask 的方式。如前所述，文档（虽然在可读取的代码中没有）提示在早期的 Android 版本中，AsyncTask 是在单线程上运行的。这个线程和 Looper 非常像。

因为 Looper 是由单线程驱动的，所以将一个要运行 30min 的任务加入队列中，会让队列中的其他工作停止 30min。可怕的 ANR（Application Not Responding，应用程序无响应）程序实际上是由停止主线程并阻碍它及时处理 MessageQueue 中的任务而引起的。

Looper/Handler 框架的语义（在单线程上的顺序执行）是在上一章讨论的。然而，它们确实意味着开发人员必须知道添加到队列中的任务状况。例如，在主线程

严禁网络传播本PDF，违者责任自负！

上翻转视图对象中的标志、计算视图对象的维度和位置，甚至是绘制大视图对象的背景都是可以合理完成的事情。网络调用、数据库查询和调整图像大小都是不可能完成的事情。

严禁网络传播本PDF，违者责任自负！

第 6 章
Service、进程和 Binder IPC

格伦道尔："我可以召唤地下的幽灵"。

霍茨波："啊！这我也会，什么人都会。可是您召唤它们的时候，它们会应召而来吗？"

莎士比亚，《亨利四世》

Android Service（服务）组件是管理并发线程的关键资源。了解它的一些细节是创建架构良好的 Android 应用程序的关键。

在应用程序中使用 Service 的原因如下。

- **架构**：从架构上来说，它使业务逻辑从表示它的 UI 中分离出来。一般来说，任何从 Activity 对象中获取代码的思路都是一个好主意。
- **进程优先级**：将一个任务委托给 Service 可以给 Service 提供宿主进程的优先级。如第 3 章中所讨论的，充当可见 Activity 宿主的进程有最高的优先级。然而通过声明框架可执行有用的工作，Service 也可以修改宿主进程的优先级。更高进程优先级会降低任务被中断的可能性，即使进程不是可见 Activity 的宿主时也是如此。
- **进程间通信**（IPC）：稍后将介绍的**绑定** Service 是 Android IPC 工具。

Service 的组件实际上是两种不同的机制，**started Service** 和封装在包中的 **bound Service**。这两种机制提供了系统可寻址的组件，它可用于执行其他线程或者其他进程中的任务。

6.1 Service 的基础知识

不管是启动型还是绑定型，这些 Service 在某些方面都是相同的。一种了解它们的

严禁网络传播本PDF，违者责任自负！

简单方式是没有 UI 的 Activity。这听起来有点奇怪，因为 Activity 的唯一目的是驱动 UI。然而，对所有服务共同特点的回顾将证明这个比喻是恰当的。

和其他 Android 组件一样，一个 Service（不管是绑定型还是启动型）必须将其组件类（这里是 **Service**）子类化。此外，和其他组件一样，一个 Service 必须在清单 6.1 所示的应用程序配置文件中注册。

清单 6.1　声明一个服务

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="net.callmeike.android.exampleservice"
  >

  <application>
    ...

    <service android:name=".ExampleService" />
  </application>

</manifest>
```

Android 框架会在安装应用程序时读取应用程序的配置文件。它注册在系统查找表中找到的组件和属性，包括 Service 在内。它们一直留在系统查找表中，直到被卸载为止。

Service 配置文件中声明的 `exported` 属性控制着它是否可见，以及如何从其他应用程序中使用它。默认情况下，服务是不可导出的，并且是外部不可见的。例如，清单 6.1 中的服务不能被其他应用程序使用。不管是声明服务 `exported="true"`，还是在声明中包含 `intent-filter`，都能使它成为外部可见的。

此外，和其他组件一样，任何时候一个服务都只有一个实例。服务的实例是在响应第一次请求它时创建的。一旦被创建，这个实例就会响应后续的所有请求（启动型或者绑定型），直到满足所有请求并且停止服务或者最后一个客户端解除绑定为止。然后，销毁服务实例。后续请求创建并使用一个新实例。

和 Activity 一样，服务有一个用回调实现的生命周期。和 Activity 不同的是（因为服务不管理 UI），其生命周期非常简单。它的生命周期状态和回调更少。`onCreate`、`onStartCommand`、`onBind` 和 `onDestroy` 是不可或缺的。

由于 `Service` 类封装两种不同的机制，因此其接口会因为接口污染而变得复杂。

严禁网络传播本PDF，违者责任自负！

它包含这两个接口的生命周期方法。虽然它可用于启动和绑定服务（一个**混合服务**），但大多数实现都是不可以的。

和 Activity 类似，服务方法总是在运行进程的主线程上运行。了解服务中长时间运行的任务会停止应用程序宿主的 UI 这一点非常重要，这与同一个任务在 Activity 中运行完全相同。使用昂贵方法的服务必须在后台进程上安排运行这些方法。

最后，和 Activity 一样，服务的生命周期影响着驱动它的进程优先级。当服务通知 Android 框架它正代表客户端执行工作时，宿主进程的优先级就高于不包含工作组件的进程。

服务优先级的细节，尤其是绑定型服务的优先级非常复杂。如图 3.2 所示，包含活动 Service 的进程优先级低于包含可见 Activity 的进程（其 `oom_adj` 值比较高），但高于只包含不可见 Activity 的进程。这是很有道理的。当一个进程包含进行有用工作的服务时，它就会比不做任何事情的进程更重要，然而，它没有用户正在查看的进程重要。

进程优先级是使用 Service 的最重要原因之一。除非长时间运行的任务是在服务中实现的，否则 Android 系统无法知道任务是否正在运行。它会降低宿主进程的优先级，甚至在任务完成前收割这个进程。服务是这个问题最好的解决方案，我们在第 4 章中详细介绍了其细节。

虽然包含运行 Service 的进程不太可能被中断，但无法保证一定会是这样的。在带有大存储器的现代设备上，一个比较小的服务，即使宣告它正在做有用的工作，也可能被中断。任务运行的时间越长，中断的可能性就越大。

6.1.1 启动型 Service 的要点

一般来说，启动型服务通常更自主。要想启动一个服务，客户端需要创建一个小参数包 Intent，然后使用 Intent 作为 **Context** 方法 `startService` 的参数。服务异步接收 Intent 作为其 `onStartCommand` 方法的参数。它执行由 Intent 表示的任务，通常不与客户有进一步的交互。和具有 `void` 返回类型的方法一样，启动型服务通常不会返回显式结果。图 6.1 演示了用 Activity 启动服务来执行一个异步任务。

注意

图 6.1 并没有显示进程边界。在讨论 Service 时，进程边界通常是不相干的。只要导出图右侧的 Service，左侧的 Activity 就会将一个 Intent 发送给它，而不管它是在一个相同还是不同的进程中。

严禁网络传播本PDF，违者责任自负！

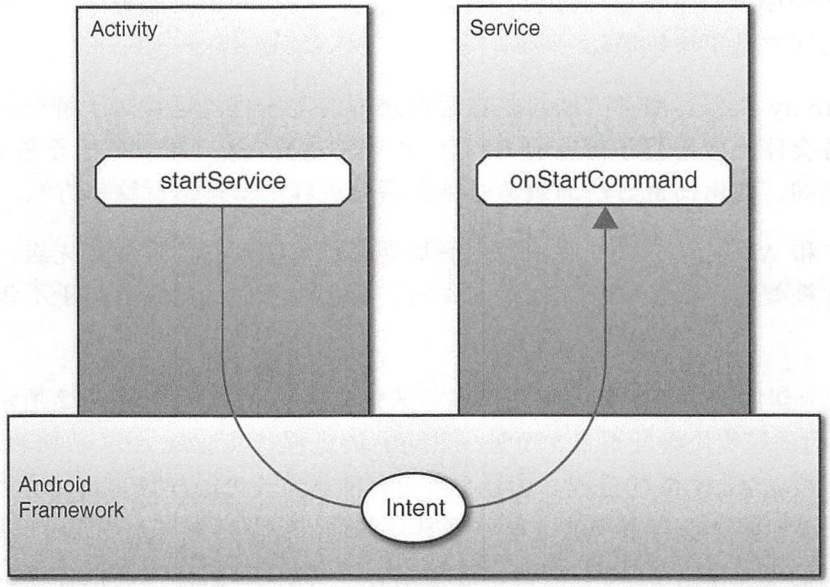


图 6.1　启动型服务

如上所述，Android 框架可以被迫中断 Service，从而为其他应用程序腾出空间。启动型服务使用它从 onStartCommand 方法中返回的值，来请求它想要 Android 框架处理这种中断的方式。这基本上有 3 种可能性。

第一种可能性是 onStartCommand 返回标志 START_NOT_STICKY。这个标志指示框架不需要采取任何附加操作。如果它必须中断启动型服务，那么服务就会标记为停止，并且框架不需要重新启动它。

而另一个极端是，onStartCommand 可能返回标志 START_REDELIVER_INTENT。这个标志指示框架如果它必须停止 Service，就应该在条件允许时重启它。

当框架中断这种 Service 时，它会将 Service 标记为停止。然而，通过重新传递与第一次启动相同的 Intent，框架可以重启 Service。Service 检查 onStartCommand 的第二个参数可以检测出它是否已经重启。如果已经重启，则 START_FLAG_REDELIVERY 就会设置。使用 START_REDELIVER_INTENT 是重启幂等任务的简单方式。

最后一种可能性稍微有点复杂。如果 onStartCommand 返回标志 START_STICKY，那么中断的 Service 就会终止，但它不会标记为停止。相反，框架会在条件允许时，通过再次调用其 onStartCommand 方法重新创建并再次运行 Service。但是，它不会再次传递原来的 Intent。重启时，onStartCommand 调用的 Intent 参数将为空。

严禁网络传播本PDF，违者责任自负！

6.1.2 绑定型 Service 的要点

图 6.2 说明了第二种服务——绑定型服务。乍一看，绑定型服务像是工厂模式的实现。

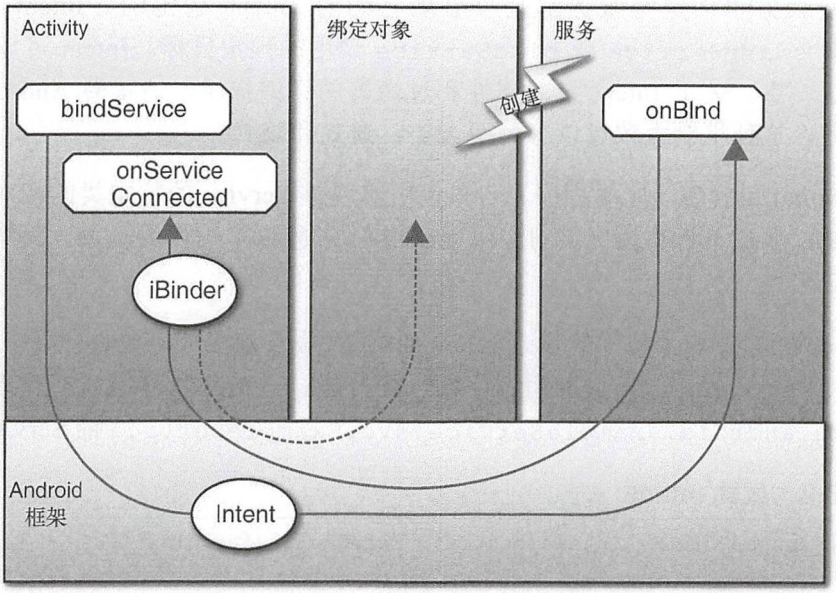


图 6.2 绑定型服务

为了绑定服务，客户端会创建一个类似于启动服务时使用的 Intent。然而，它调用的是 bindService，而不是 startService。除了 Intent，它还会提供对回调处理函数的引用，即 ServiceConnection 接口的实现。

Service 异步接收 Intent，这次它作为参数的调用其 onBind 方法。作为对 onBind 的响应，它返回所托管对象的实例。

Android 框架再次异步调用回调处理函数的 onConnected 方法，它是由客户端传递的。作为调用参数，它会传递一个对托管对象的引用，和从 onBind 方法中返回相类似。现在，客户端有一个由 Service 提供的对象实例，并且它可以用与其他 Java 对象交互时类似的方式和它进行交互。

作为一个工厂，绑定型服务在返回给客户端的对象类型和实际实现之间提供了一个抽象级别。在第二个抽象层，是由 Android 框架收集 onBind 调用的结果，然后将

严禁网络传播本PDF，违者责任自负！

结果（或者类似的东西）传递给 onConnected 调用。这对于绑定型服务，即进程间通信的使用是非常重要的，其中对象的实现可以处于不同的进程中。

6.2 Intent

启动型服务和绑定型服务都是由启动它们的 Intent 选定的。Intent 必须识别 Service，即 startService 或 bindService 调用的预定目标。Intent 可以通过集中方式做到这一点。关于 Intent 及其属性和过滤器的详细描述，请参阅 Android 的相关文档。然而，这种机制大致可以分为两大类：**显式**和**隐式**。

显式 Intent 包含包（应用程序）名和包中实现该 Service 的特定类的相对名称。最好的情况是，显式 Intent 在特定的应用程序中准确地命名一个 Service。最坏情况是，它谁都不命名。

清单 6.2 展示了 3 个相同的显式 Intent 的构造（假设这段代码是 net.callmeike.android.exampleService 应用程序的一个片段，该应用程序包含一个名为 net.callmeike.android.exampleService.svc.ExampleService 的 Service）。

清单 6.2　显式 Intent

```
Intent intent1 = new Intent(context, ExampleService.class);

Intent intent2 = new Intent();
intent2.setComponent(new ComponentName(
  context.getPackageName(),
  ExampleService.class.getName());

Intent intent3 = new Intent();
intent2.setComponent(new ComponentName(
  "net.callmeike.android.exampleservice",
  "net.callmeike.android.exampleservice.svc.ExampleService");
```

重要的是，要注意显式 Intent 不保留对类对象（ExampleService.class）或者传递给它的 Context 对象的引用。虽然清单 6.2 所示的最简单结构（即用于创建 intent1 的结构）使用了这两个参数，但它只是为了创建名字字符串。正如 intent3 的创建所示，在其他应用程序的名称组件中创建显式 Intent 是完全可能的（假设这些组件名是已知的）。

严禁网络传播本PDF，违者责任自负！

隐式 Intent 是必须使用**意图过滤器**（intent filter）捕获的 Intent。IntentFilter 是用于匹配标准的 Intent 属性：action、package、category、MIME type 等的 Android 框架对象。清单 6.3 所示的 IntentFilter 是为 Service 创建的，它在其应用程序配置文件中是注册 Service 的一部分。

当 Service 启动一个 IntentFilter 时，它就成为接收匹配过滤器规范的 Intent 的候选。在清单 6.3 中，注册服务 `ExampleService` 来接收其 `action` 属性为 `net.callmeike.android.exampleService.PING` 的 Intent。

清单 6.3 IntentFilter

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="net.callmeike.android.exampleservice"
  >

  <application>
    ...

    <service android:name=".ExampleService">
      <intent-filter>
        <action android:name="net.callmeike.android.exampleservice.PING" />
      </intent-filter>
    </service>
  </application>

</manifest>
```

正如大多数 Android 用户所知道的，不明确的 Activity 调用（匹配多个 Activity 的对 Intent `startActivity` 的调用）是通过向用户提供系统对话框并允许他从中选择来解决的。例如，考虑当设备中安装着多个浏览器或者多个电话应用程序时会发生什么。

此外，将 Intent 传递给 Service 的消歧协议是沉默的，本质上是随机的。因此，使用隐式 Intent 启动 Service 是非常不安全的。虽然从最初就是不安全的，但是从 API 21 KitKat 开始，用于调用 `startService` 的 Intent 才必须显式命名一个目标应用程序。

在写本书时，Android 文档要求用于标识 Service 的 Intent 必须是显式 Intent。事

实上，这是不必要的。Intent 通过指定其包名来指定一个唯一的目标应用程序就足够了。在最低程度下，使用具有 `String` 参数的 Intent 方法 `setPackage` 是可以做到的，其中这个参数有资格指定目标应用程序的包名。

显然，这仍然可能留下不明确的 Service。然而，应用程序中不明确的 Service 只是一个缺陷，而不是一个安全问题。

在并发背景下要了解的一点是，客户端用于调用 Service 的 Intent 永远不会是被 Service 调用而接收的 Intent。尤其是，清单 6.4 所示的代码总是产生如下所示的输出：

```
...
01-10 16:58:51.102 1953-1953/…servicesandbox I/SVC: intent == originalIntent:
false
...
```

清单 6.4　Intent 识别

```
public class SimpleService extends Service {
  // ...

private static Intent startIntent;

@UiThread
public static void start(Context ctxt) {
  if (null != startIntent) {
    throw new IllegalStateException("start called twice");
  }
  startIntent = new Intent(ctxt, SimpleService.class);
  ctxt.startService(startIntent);
}

// ...

@UiThread
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
  Log.i(TAG, "intent == originalIntent: " + (intent == startIntent));
  startIntent = null;
    startWork();
```

```
        return super.onStartCommand(intent, flags, startId);
    }

    // ...
}
```

这有两个结果。第一个是 Service 不必关心是否有传递给它的额外的 Intent 引用。Service 接收到的 Intent 是客户端在调用 `startService` 或 `bindService` 时使用的 Intent 的深拷贝。Service 持有其引用的是 Intent 其本身，Service 可以在没有线程安全问题的情况下使用它。

第二个结果是，Intent 必须是 Android 系统可以进行深拷贝的对象。Java 中的 `Cloneable` 接口表示对象可以被复制的标准方式。Android Intent 是 `Cloneable`，但它们实际上支持更强的约定 `Parcelable`。支持 `Parcelable` 约定的对象可以在 Android 进程之间传输。

可以在进程之间传输的对象的 Java 约定是 `Serializable` 接口。一个可序列化对象可以在其他地方和时间，通过一个名为**封送**（marshaling）的进程进行存储重构。对象的封送表示在它的新环境中可以是不分组的，以创建一个新的不同对象，但它实际上与原来的对象相同。

下面将详细讨论的 `Parcelable` 接口是 Android 中 Java `Serializable` 的轻量级等效。显然，可被封送和不可封送的对象都可以备份。然而，由于它们都支持 `Parcelable` 约定，因此 Intent 可以跨进程边界传递。

6.3 IntentService

`IntentService` 是使用启动型服务最常见的方式。它是基础 Service 类的简单、美观和优雅的延伸，为开发人员提供了转向 AsyncTask 时需要查找的功能。

回顾上一节，除非 Service 的实现专门安排了替换方法，否则它的所有代码都在 UI 线程上运行。这使它在实现长时间运行的任务时变得非常没用。再说一次，问题是从 UI 线程获取任务。

第 3 章讨论了用于并发的 Android naïve 工具（AsyncTask），并且描述了在 Activity 上下文中使用它们会出现的很多问题。然而，在 Service 上下文中重新考虑它们很有帮助。

清单 6.5 显示了执行任务的 Service，它是通过 onStartCommand 调用，在后台线程上实现的。

清单 6.5　AsyncTask Service

```
public class AsyncTaskService extends Service {
  private int running;

  // ...

  @UiThread
  public int onStartCommand(Intent intent) {
    running++;
    new AsyncTask<Intent, Void, Void>() {
      @UiThread protected void onPostExecute(Void result) { finishTask(); }
      @UiThread protected void onCancelled(Void result) { finishTask(); }
      @WorkerThread protected Void doInBackground(Intent... intent) {
        executeTask(intent[0]);
        return null;
      }
    }.execute(intent);
    return Service.START_NOT_STICKY;
  }

  @UiThread
  void finishTask() {
    if (0 >= --running) { stopSelf(); }
  }

  @WorkerThread
  void executeTask(Intent intent) {
    // do task work...
  }
}
```

首先，没有什么可以约束 AsyncTask 只能在 Activity 中使用。它们在其他组件中也能运行良好。

清单 6.5 中的 AsyncTask 是后台线程的门户。当客户端调用 startService 方法

时，参数 Intent 在调用 onStartCommand 时传递给 Service。在 AsyncTaskService 中实现的 onStartCommand 立即将 Intet 转发给一个新的、匿名的 AsyncTask 实例，然后实现这个实例。

如第 3 章所述，这会导致 AsyncTask 方法 doInBackground 在后台线程中运行。在 AsyncTaskService 中，匿名 AsyncTask 的 doInBackground 方法会调用 Service 方法 executeTask。

这实现了什么？现在 Service 方法 executeTask 几乎与 onStartCommand 相同，除了它在 AsyncTask 的工作者线程上运行。它现在是实现长时间运行任务的好地方。

情况越来越好了。调用 Service 方法 onStartCommand 时有一个副作用：它会导致框架将 Service 标记为“启动的”。它向 Android 宣告，这个进程正在进行有用的工作。作为响应，框架会让启动型服务的宿主进程优先级高于空闲进程的优先级。因此，与在具有不可见 Activity 的进程中运行同一个任务相比，在这个 Service 上运行的 AsyncTask 更有机会运行完成。

Android 框架能很容易地判断出何时不再需要 Activity。当它不再可见时，它就不再做有用的工作了。没有人知道启动型线程不再有用的类似方法。一旦 Service 标记为启动，它就会保持“启动”，直到它被显式停止为止。为了成为一个更好的邻居（在不需要时释放资源），Service 的客户端必须调用 stopService 或者 Service 必须调用它的方法 stopSelf。这些调用向框架宣告 Service 是通过有用的工作来完成的，从而告诉框架它可以减低宿主进程的优先级。

AsyncTaskService 通过维持运行任务数来管理其状态。它在新 Intent 进入 onStartCommand 时递增；当任务完成时，在 AsyncTask 的方法 onPostExecute 中再次递减它。由于这两个方法都在主线程上运行，因此不需要同步访问计数器。当计数器达到 0 时，Service 调用 stopSelf 向框架声明它不再做有用的工作了，这个实例可以被销毁，并且宿主线程的优先级会降低。

注意

清单 6.3 及围绕它进行的讨论仅作为示例。这些代码是不完整的。例如，它算错了已终止的任务数。

AsyncTaskService 具有我们一直在后台执行机制中寻找的特性。它在工作者线程上运行任务，并调整进程优先级来降低 Android 框架中断后台工作的机会。此外，在被 Service 孤立的环境中，远离 UI 机器的地方不太容易意外地跨线程共

享状态。

提供所有特性的 Service 似乎是很有价值的东西。Android 的设计者已经意识到这种工具的重要性，因而在 API level 3 Cupcake 中，将 `IntentService`（具有相似功能的服务）引入 Android 库中。

IntentService 的行为几乎和 `AsyncTaskService` 一样，但其实现稍微有点不同。它在后台线程上使用标准方法 `onHandleIntent` 来执行任务。和 AsyncTaskService 一样，它宣称自己不粘贴，这样 Intent 只传递给 onHandleIntent 一次并且不为空。一个非常明显的差别是 IntentService 由私有 Looper 和其工作者线程驱动，而不是由 AsyncTask 和备份它们的 `ThreadPoolExecutor` 来驱动。

这会产生几种后果。首先，在 IntentService 的工作者线程上运行的任务可以共享状态。由于它们都保证在同一个线程上运行，因此线程之间不共享任何状态。显然，在主线程上运行的 Service 生命周期方法（`onCreate`、`onStartCommand`、`onDestroy` 等）必须小心，不要与任务共享状态。然而，任务本身可以彼此共享状态。

然而，更重要的是，任务在 IntentService 中按顺序执行。当每个新的 Intent 出现在 `onStartCommand` 调用中时，它就会添加到 Looper 的消息队列中，最终按顺序出队，并在后台线程上表示为 `onHandleIntent` 方法的参数。只有当这个方法让 Intent 完成并返回时，才会为了后续的 onHandleIntent 调用让下一个 Intent 出队。

正如在 AsyncTask 的讨论中所示，顺序执行通常是有优势的。对顺序任务组的推理要比对异步任务组的推理简单得多。例如，从概念上来说，只是因为网络任务可能不按顺序发生，所以在 UI 任务方面，网络和 UI 任务就不需要按顺序发生。实际上，让用户登录账户的网络任务必须在下载其数据任务之前发生。允许这些事情不按顺序发生并不一定有优势。

为不同任务类使用多个 IntentService 且彼此异步但内部有序的架构很有意义。然而，对需要完全异步任务执行的架构来说，IntentService 不是正确的工具。然而，基于 `AsyncTaskService` 建模的自定义 Service 可能可以完成任务。

通过作为 Service 类一部分的静态方法 IntentService 经常被调用。这些方法通常用 `Context` 和需要传递给任务的任何参数作为参数。它们将参数封送到 Intent 中，然后调用 `startService` 传递新的 Intent。常见的 IDE（Eclipse 和 Android Studio）从代码模板中生成这个习惯用法。静态方法从客户端隐藏 Intent 结构的细节，并提供一种相当优雅的调用 Service 功能的方式——像任何静态引用一样优雅。清单 6.6 所示为这种辅助方法的示例。

版权所有，严禁传播，违者自负法律责任！

清单 6.6 Intent Service 的辅助方法

```
public class CookieService extends IntentService {
  // ...

  private static final String ACTION_EAT
    = "net.callmeike.android.samplesvc.svc.action.EAT";
  private static final String EXTRA_PARAM_COOKIE
    = "net.callmeike.android.samplesvc.svc.extra.COOKIE";

  // ...

  public static void startActionDoIt(Context ctxt, String cookie) {
    Intent intent = new Intent(context, SampleService.class);
    intent.setAction(ACTION_EAT);
    intent.putExtra(EXTRA_PARAM_COOKIE, cookie);
    context.startService(intent);
  }

  // ...
}
```

IntentService 最重要的约束可能是让它返回结果如何困难。虽然启动 Service 的 Intent 可以在某些限制条件下携带任务参数，但 IntentService 如何返回结果并不是显而易见的。当然，典型的 Android 解决方案是某种回调。在这种情况下，通过使用 Handler 将结果传送回主线程，然后在从这里传递回调处理函数中的调用方法即可完成。在稍后讨论 **messenger** 时也要用到这种想法。

6.4 绑定型 Service

绑定型服务有几个很吸引人的特性。在一个进程中，调用由绑定型服务管理的对象要比将 Intent 发送给 Intent Service 速度快几个数量级。此外，有了绑定型服务，更容易创建双向 API，因为方法在其中返回值。然而，它们最重要的特性是 Service 返回的托管对象不仅可以在其他线程上运行，还可以在其他进程上运行。

绑定型服务是 Android 应用程序开发人员遇到的最复杂的事情之一。我们暂时先跳过这种复杂性，探讨一个非常简单的绑定型服务客户端。

版权所有，严禁传播，违者自负法律责任！

如前所述，绑定型服务初看起来就像托管对象的工厂。然而，仔细检查就会发现这个模式只是部分准确。

6.4.1　一个简单的绑定型 Service

回想图 6.2，客户端通过调用 Context 方法 `bindService` 初始化绑定进程。在这个调用中，它传递识别目标 Service 的 Intent，以及对回调处理函数（`ServiceConnection` 实例）的引用。它还传递一些标志（稍后讨论）。代码如清单 6.7 所示。

清单 6.7　简单的绑定型服务客户端

```
public class SimpleActivity
  extends AppCompatActivity
  implements ServiceConnection
{
  @Override
  public void onServiceConnected(ComponentName name, IBinder binder) {
    setService(((LocalService.ServiceBinder) binder).getService());
  }

  @Override
  public void onServiceDisconnected(ComponentName name) {
    setService(null);
  }

  @Override
  protected void onStart() {
    super.onStart();
    bindService(
      new Intent(this, LocalService.class),
      this,
      BIND_AUTO_CREATE);
  }

  @Override
  protected void onStop() {
    super.onStop();
```

版权所有，严禁传播，违者自负法律责任！


```
    setService(null);
    unbindService(this);
  }

  // ...

  private void setService(LocalService svc) {
    service = svc;
    addButton.setEnabled(svc != null);
  }
}
```

如果 Android 框架不能成功地将参数 Intent 传递给某个 Service，那么调用 bindService 就会返回一个布尔值 `false`。发生这种情况的原因是 Intent 无法匹配系统注册的 Service，或者系统不能启动包含已注册 Service 的应用程序。如果 `bindService` 返回 `false`，绑定进程就会中止。

如果系统成功定位了一个可绑定的服务，那么调用 `bindService` 就会返回 `true`。目标 Service 异步接收 `bindService` 调用发送的 Intent 备份，作为调用其 `onBind` 方法的参数。`onBind` 方法最终返回一个对象，而 Android 再次异步传递它（或者，至少是非常类似的东西），并且作为调用回调处理函数的 `onServiceConnected` 方法的参数（`bindService` 调用的第二个参数）。

和在 IntentService 中一样，绑定服务的应用程序负责释放未使用的资源，并解除对不需要的 Service 的绑定。客户端通过调用 `Context` 方法 `unbindService` 并传递绑定它的回调处理函数，来解除对 Service 的绑定。如果它是解除绑定的最后一个客户端（如果没有其他绑定的客户端），框架就会调用 `Service` 的 `onUnbind` 方法。

6.4.2 绑定一个 Service

客户端代码必须能够构造一个 Intent，来命名想要作为目标的 Service。如前所述，要发布 Marshmallow API 23，客户端代码必须知道包含 Service 应用程序的包名。此外，它必须知道如何匹配目标 Service 的 IntentFilter。如果目标 Service 还没有声明 IntentFilter，那么客户端就必须知道实现 Service 的类名。

为了支持这一需求，很多为公众创建 Service 的开发人员在一个名为 `Connected` 的文件中记录了 Service API。约定是一个 Java 源文件，为使用 Service 所需的常量定

版权所有，严禁传播，违者自负法律责任！

义符号。文件必须不能包含对应用程序内部符号或者类型的引用，它可以独立分布，也可以是作为网站下载或者通过 E-mail 来分布。

客户端应用程序在它们的源文件中包含约定文件，并使用定义的符号（甚至是它提供的文档）正确地与公共 Service 交互。Android 文档建议使用约定将接口发布到 ContentProvider 中。它们还可以发布可绑定的 Service API。

“约定是个好主意”，还有更重要的原因。要注意的是，`onServiceConnected` 方法（调用它来将 Service 管理的对象传递给客户端）将类型 `IBinder` 的对象作为参数。这是 Service API 和标准工厂模式分歧的开始。

和工厂模式中大多数 Java 实现不同，Service 既不是同步的，也不是类型安全的。客户端接收到 Service 管理的对象实例作为回调，而不是调用工厂方法的返回。此外，Service `onBind` 方法返回一个 `Binder` 对象，而客户端接收到一个 `IBinder`。客户端的责任是要知道期待什么类型的对象，并将它接收到的 `IBinder` 类型转换为正确的 Java 类型。

“Service 作为工厂”这个隐喻的改进是由调用的 Service `onBinder` 方式提出的。在创建给定 Service 实例和销毁它之间，其 `onBind` 方法只被所有 Intent 调用一次。

例如，所有想将给定 Service 作为目标的显式 Intent 都是相同的——它们都包含同一个包的名字和这个包中同一个类的名字。由于它们都是相同的，因此只有第一次尝试用显式 Intent 绑定 Service 时会产生对 Service `onBind` 方法的调用。

对 `bindService` 的后续调用不会导致再次调用 `onBind`。相反，Android 框架只是返回第一次调用 `onBind` 时获取的对象。Service 也许更像是单例模式，而不是工厂模式。

注意

21 世纪在企业软件中常见的架构反模式会涉及用大型单例模式建立的系统，通常称为“管理器”。这些系统在名义上是整体的、面向对象的。它们难以测试，并且舍弃了面向对象编程可以提供的优点。

Steve Yegge 在他的博客上写了一篇关于单例模式和架构的文章。

如果你是那个时代的开发人员，那么对 Android Service 的讨论可能会让你在架构方面觉得不舒服。

版权所有，严禁传播，违者自负法律责任！

6.4.3 解除对 Service 的绑定

然而，服务作为单例并不成立。在发现 unbindService 方法时它会崩溃。通常单例并不受到破坏。unbindService 方法的存在很令人好奇，需要仔细检查。

首先，要注意 bindService 和 unbindService 是类型 Context 实例的状态方法。当一个服务被绑定时，绑定的 Context 就会记住这个绑定。只有这个 Context 可以解除连接的绑定。例如，清单 6.8 所示的代码会产生如下错误：

```
...
java.lang.IllegalArgumentException: Service not registered:
  net.callmeike.android.servicedemo.MainActivity@6ccc2eb
...
```

清单 6.8 从错误的 Context 解除绑定

```
@Override
protected void onStart() {
  super.onStart();
  bindService(new Intent(this, LocalService.class), con, BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
  super.onStop();
  getApplicationContext().unbindService(con); // !!! ERROR!
}
```

问题是清单 6.8 中的 bindService 方法是在 Activity（即上下文）中调用的。然而，unbindService 方法是在应用程序上下文上调用的。应用程序上下文当然是一个 Context，但不是和 Activity 一样的对象。应用程序上下文没有连接记录，因此尝试解除绑定会失败。

Activity 绑定 Service 时会产生一个有趣的推论。由于绑定属于特定的 Context（Activity），并且框架控制着这个 Context 的生命周期，因此它知道当销毁 Activity 时，任何还没有被绑定的绑定都不能被正确地绑定。若在没有将已绑定的 Service 解除绑定的情况下将 Activity 销毁，框架就会产生如下所示的错误消息：

```
...
```

版权所有，严禁传播，违者自负法律责任！


```
Activity net.callmeike.android.servicesandbox.MainActivity has leaked
  ServiceConnection net.callmeike.android.servicesandbox.MainActivity@1d77f05
  that was originally bound here
...
```

绑定一个 Service 会影响该 Service 宿主进程的优先级。框架期望绑定 Service 的组件显式地管理该绑定，并在不需要它时解除绑定。如果没有，框架就会发出错误信号。

然而，关于 unbindService 方法最重要的是，它可能不会立即对在 onService-Connected 方法中将其引用传递给客户端的对象产生影响。可以合理地期望一个 Service（尤其是如果这个 Service 在远程进程中），它会将连接提供给远程对象。但是它不会。

这就导致绑定型服务的另一个类比。绑定型进程类似于单例中的 Java SoftReference。当绑定服务时，框架就会承诺在客户端用修改引用时提醒它。当它将引用设置为指向有用的对象时，就会调用 onServiceConnected。当它将引用设置为空时，就会调用 onServiceDisconnected。

然而，当 Service 不再绑定到一个回调处理函数时，框架就没有义务通知任何人了。但 Service 提供的引用不会变成无效的。然而，框架可以在没有通知客户端的情况下让它成为无效的。

当没有客户端监听 Service 状态的修改时，框架就会假设没有人关心它。当客户端正在监听状态的修改时，框架就会假设 Service 正在为它们工作，并且只有在必要时这个 Service 才会销毁。如果它必须销毁 Service，那么它会通知客户端，并尝试尽快重新创建 Service。然而，正如在没有人能看到 Activity 时才会考虑它，只有在没有绑定时才会考虑绑定型服务。Service 对象可以被销毁，主机优先级可以被降低。

6.4.4　绑定多个 Service

和 Android 中的很多其他回调处理函数一样（View.onClickHandler、LoaderManager. LoaderCallbacks），ServiceConnection 只能管理一个绑定。将绑定 Service 的上下文（通常是 Activity）作为回调处理函数是一个常见的习惯用法，如清单 6.7 所示。然而，它是只允许绑定一个服务连接的习惯用法。若要连接到多个 Service，则需要多个回调处理函数，如清单 6.9 所示。

版权所有，严禁传播，违者自负法律责任！

清单 6.9 将多个服务绑定到一个 Context 中

```
public class Bind2Activity extends AppCompatActivity {
  private ServiceConnection con1 = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder binder) {
      svc1 = (((LocalService1.ServiceBinder) binder).getService());
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
      svc1 = null;
    }
  }

  private ServiceConnection con2 = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder binder) {
      svc2 = (((LocalService2.ServiceBinder) binder).getService());
    }

  @Override
  public void onServiceDisconnected(ComponentName name) {
    svc2 = null;
  }
}

// ...

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main2);

  bindService(
    new Intent(this, LocalService1.class),
    con1,
```

版权所有，严禁传播，违者自负法律责任！


```
      BIND_AUTO_CREATE)
    bindService(
      new Intent(this, LocalService2.class),
      con2,
      BIND_AUTO_CREATE)
    }

    // ...
}
```

6.4.5　Service 的生命周期

当意识到 Service 需要客户端精心合作来防止获取它所管理对象的多个副本时，这就是 Service 作为单例隐喻的终结。

对于每个等效的 Intent，在 Service 实例的生命周期期间，Android 框架只调用一次 Service 的 `onBind` 方法。此外，从其客户端的角度来看，Service 对象的生命周期几乎是随机的。单个 Service 的不同实例可以轻松地返回不同的托管对象。

考虑清单 6.10 所示的最小 Service。对于每个 `onBind` 调用来说，`MinimalService` 返回其管理对象 `ManagedObject` 的一个新实例。由于对绑定 `MinimalService` 的显式 Intent 来说只调用一次 `onBind`，因此人们可能会认为 `ManagedObject` 是一个单例。

清单 6.10　最低程度的绑定型服务

```
public class MinimalService extends Service {

  @Override
  public IBinder onBind(Intent intent) {
    return new ManagedObject();
  }
}
```

其实并不是这样。如图 6.3 所示，一个客户端很容易就可以结束对托管对象两个不同实例的引用。

图 6.3 所示的客户端对 `bindService` 的第二次调用会转发给 Service 的一个不同实例。框架没有托管对象的缓存实例（当销毁 Service 时它被清除），因此它会在 `onBind`

版权所有，严禁传播，违者自负法律责任！

中调用。

onBind 方法创建一个新对象并返回它。在解除 Service 绑定时没有错误地忘记托管对象第一个实例的客户端，现在接收到第二个实例的引用。显然，客户端代码有一个实例。然而，现在它有两个不同“单例”的引用。

此外，还有生命周期。即使修复了清单 6.10 所示的代码，从而可以让托管对象成为一个单例（见清单 6.11），但与单例类似的行为仍然可能导致意外的发生。

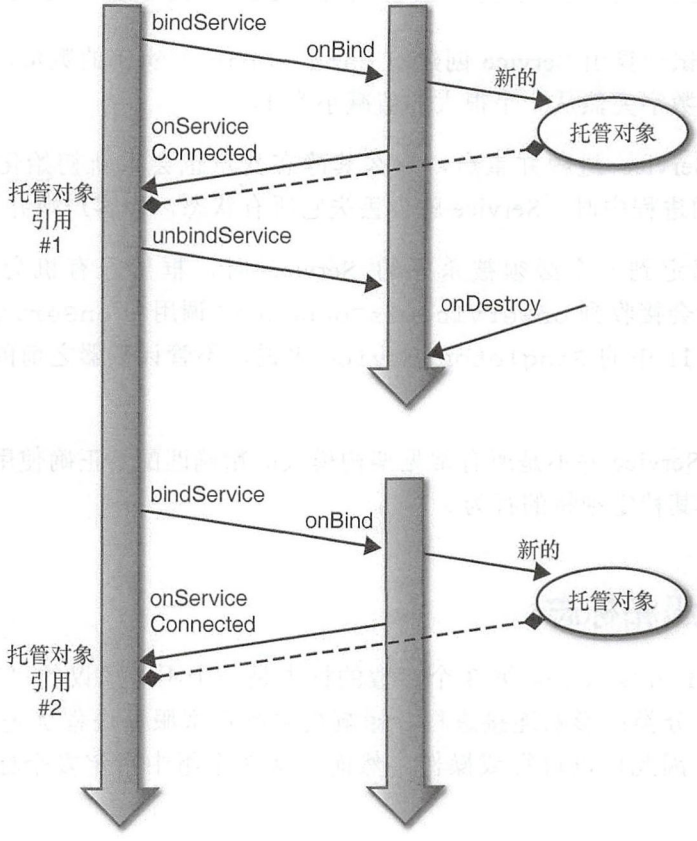


图 6.3 Service 管理的对象不是单例

清单 6.11 单例绑定型服务

```
public class SingletonService extends Service {
  private static ManagedObject managedObject;
  private static int counter;
```

```
    @Override
    public IBinder onBind(Intent intent) {
      if (null == managedObject) {
        managedObject = new ManagedObject(++counter);
      }
      return managedObject;
    }
  }
```

这段代码尝试计算由 Service 创建的 ManageObject 实例的数量。然而，客户端可能会看到这个数字突然从一个很大的值减小为 1。

如果停止 Service 进程并重启，那么其静态变量也会重新初始化。当客户端和 Service 在不同的进程中时，Service 就会丢失它所有状态，但客户端并不会。

当客户端绑定到一个必须被杀死的 Service 时，框架一有机会就会重启这个 Service。客户端会接收到 onServiceDisConnected 调用和 onServiceConnected 调用。对清单 6.11 中的 SingletonService 来说，不管计数器之前的值是多少，它现在都是 1。

事实证明，Service 并不是所有常见架构模式的精确匹配。正确使用它们需要理解它们，以及理解其特定独特的行为。

6.4.6　优先级和标志

作为调用 bindService 第 3 个参数的标志是一个不可思议的“毛球”。它们几乎都属于这 3 种分类：影响连接进程、影响宿主绑定型服务进程优先级和 WTF。标志是一个位组，因此可以进行或操作。然而，从 3 个组中指定多个标志可能没有太大意义。

在第一组分类中，影响连接进程的标志是最重要的标志 BIND_AUTO_CREATE。绝大多数 bindService 调用会指定这个标志。

当指定这个标志时，它就会告诉框架以启动还没有运行的宿主进程，并创建 Service 类的一个实例（如果不存在），调用目前正在运行的实例的 onBind 方法。

当考虑不使用 BIND_AUTO_CREATE 标志调用 bindService 时，会发生什么。即使调用 bindService 返回 true（这表明框架已经成功地将 Service 定位到要绑定的

地方），也可能没有对回调处理函数的后续调用。如果没有调用回调处理函数，那么可能会经过一些时间，这个时间无法确定。

假设由于某种原因需要几个小时，则其他代码就会用指向同一个 Service 的 Intent 调用 `startService`。由于会创建并启动一个 Service 实例作为对这个调用的响应，在它调用 `bindService` 的时间内，绑定客户端的回调处理函数会突然接收到连接。很奇怪的一点是，这似乎是不可避免的，有人会发现它是某个问题的完美解决方案。

在 WTF 分类中是 `BIND_DEBUG_UNBIND` 标志。

剩余的标志 `BIND_NOT_FOREGROUND`、`BIND_ABOVE_CLIENT`、`BIND_ALLOW_OOM_MANAGEMENT`、`BIND_WAIVE_PRIORITY`、`BIND_IMPORTANT` 和 `BIND_ADJUST_WITH_ACTIVITY` 都控制着绑定型服务宿主进程的优先级。除了 `BIND_NOT_FOREGROUND` 外，它们都在 API 14（Ice Cream Sandwich）中引入。虽然相关文档详细地描述了它们，但它们的确切含义很复杂，很难透彻理解。

6.4.7 本地绑定型 Service

在讨论进程间通信之前，还有一个主题要处理。在讨论 IPC 前引入了对进程间通信的优化，这可能有点不合理。然而，它是很重要的优化，在这里介绍它就不会中断对 IPC 的讨论了。

本章在很大程度上忽略了进程边界，所有插图都不包含它们，所有代码都没有提及它们。对于大多数应用程序，甚至是大多数 Android 应用程序中的代码来说，显而易见的假设是，所有内容都是在一个线程中运行的。不管 Service 及其客户端是否在相同的进程中，本章的大多数讨论和代码都适用。Service 对进程边界几乎是视而不见的。

然而，稍微考虑一下，已经知道同一个进程将 Service 作为其客户端。在受限的情况下，有一个技巧可以消除由 IPC 框架强加给 Service 托管对象的大部分通信开销。让人遗憾的一点是，Service（模糊进程间通信和进程外通信的一种工具）通常以某种方式限制在一个进程中。然而，这个技巧对执行有巨大的影响，以至于开发人员在进程中不会使用它。清单 6.12 演示了这个技巧。

清单 6.12 本地 Service

```
public class LocalService extends Service {
  public class ServiceBinder extends Binder {
```

非卖品，仅供非商业用途或交流学习使用

```
    public LocalService getService() { return LocalService.this; }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return new ServiceBinder();
  }

  public Foo doSomething(Bar bar) {
    // ...
  }
}
```

内部类 ServiceBinder 有两个基本属性。由于它扩展了 Binder，因此可以实现 IBinder 接口，并且可以由 onBind 方法返回。Binder 是一个实现了 IBinder 并且有很多超能力的大重量级对象，然而在特殊情况下，这些能力都是无关紧要的。由于 Service 和其客户端在同一个进程中，因此客户端回调处理函数会接收 onBind 返回的对象作为其 onServiceConnected 方法的参数。要注意的是，当客户端和 Service 在不同的进程中时，情况并非如此。

由于 ServiceBinder 是 LocalService 的内部类，因此它持有对 Service 的引用。这是 Java，它会公开引用，并且在其 getService 方法中这么操作。当客户端接收回调时，就会将传递的 IBinder 参数作为 LocalService.ServiceBinder，然后调用 getService 方法。现在，它持有对 LocalService 对象的引用。清单 6.13 演示了 LocalService 客户端。

清单 6.13　本地 Service 客户端

```
private LocalService service;

// ...

@Override
public void onServiceConnected(ComponentName name, IBinder binder) {
  service = ((LocalService.ServiceBinder) binder).getService();
}

// ...
```

非卖品，仅供非商业用途或交流学习使用

这是一种习惯用法。在最常见的表示中，在 onBind 和 onServiceConnected 方法之间传递的 Binder 会持有对 Service 本身的引用，如清单 6.12 所示。这只是因为这样做很容易。Service 不只是托管对象的工厂，还是工厂返回的托管对象。

这是不必要的，因为它会使测试变得相当困难。例如，在 LocalService 上为客户端可见的方法构造单元测试时需要模拟 Service。这会使它除了在可以实例化 Service 的环境中，其他地方都很难运行测试，这包括 Android 设备或者 Robolectric。

然而，ServiceBinder 可以返回它想要的任何对象。更合理的解释是，ServiceBinder 的 getService 方法返回对一个对象的引用，而这个对象已经通过使用 IOC 框架注入 Service 中了。清单 6.14 是这种技术使用 Dagger2 的一个示例。

清单 6.14 Service 注入

```
public class LocalService extends Service {
  public class ServiceBinder extends Binder {
    private final ServiceProvider provider;
    ServiceBinder(ServiceProvider provider) { this.provider = provider; }
    public ServiceProvider getService() { return provider; }
  }

  private static ServiceBinder serviceBinder;

  @Inject ServiceProvider provider;

  @Override
  public void onCreate() {
    super.onCreate();
    if (serviceBinder == null) {
      DaggerProviderComponent.create().inject(this);
      serviceBinder = new ServiceBinder(provider);
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return serviceBinder;
  }
}
```

非卖品，仅供非商业用途或交流学习使用

6.5　进程间通信

本章已经多次提到一个事实，即 Service 可以在与客户端完全不同的进程中运行，而这个 Service 是进程间通信的 Android 工具。对 Service 的讨论是理解如何在一个不同的进程（而不只是线程）中执行任务的基础。

进程间通信的 Java 标准是 `Serializable` 接口。Android 支持 Java 序列化，但为必须序列化的对象引入了一种新的又比较简单的约定 `Parcelable`。除了 `Parcelable` 接口外，Android 工具套件还包含一个 Android 进程间通信语言——编译器，其中这种语言是 Android 接口定义语言（AIDL，Android Interface Definition Language）。

图 6.4 说明了 Android IPC 机制。

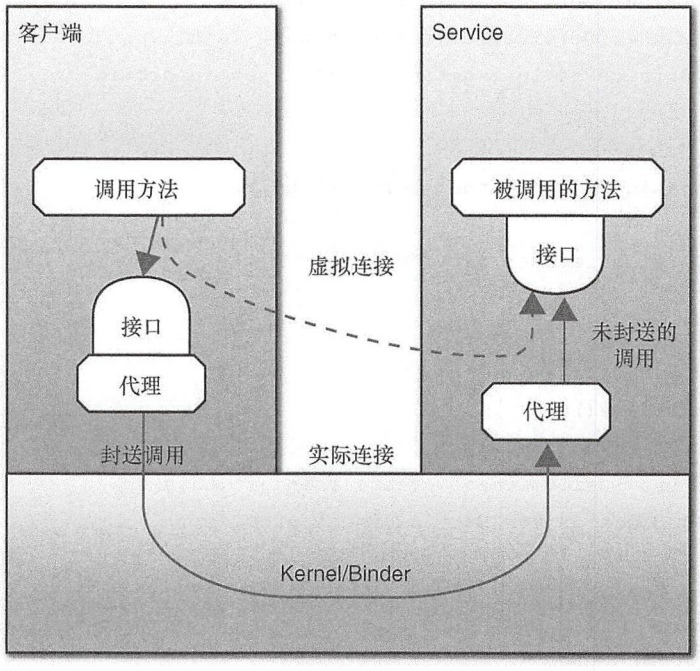


图 6.4　Binder IPC

和大多数 IPC 机制一样，Android 的 IPC 从概念上分为两部分：客户端和服务器端。这两端分别称为**代理**和**存根**。客户端认为它正在与某个对象的实例进行对话。它从工厂获取这个对象，然后工厂悄悄地返回所需对象的代理，而不是其实例。代理看

非卖品，仅供非商业用途或交流学习使用

起来非常像远程对象。在 Java 的术语中，这可能意味着它实现了相同的接口。

代理的职责是将调用封送到消息中，而不是真正地包含实现代码。消息传输给远程进程。在 Android 中，这种传输由 Linux 内核中的一个模块（称为 Binder）来处理。

Binder 将封送的消息复制到目标进程的内存空间中，并在那里将它递交给存根。存根的工作是取消消息的封送，并调用对象的实际实现：客户端认为它正在调用的对象。

如果调用返回一个值，那么进程就会反过来将返回值传输回客户端。

6.5.1 Parcelable

一个 Parcelable 对象是实现特定 API 的对象。首先，它必须实现 `Parcelable` 接口。这个接口需要实现两个方法：`describeContents` 和 `writeToParcel`。此外，Parcelable 对象必须确实有一个公有静态成员 `CREATOR`，它是 Parcelable 类型的 `Parcelable.Creator` 实现。

`writeToParcel` 方法 `CREATOR` 分别是 marshaler（封送处理器）和 unmarshaler（拆封处理器）。

当框架需要将 Parcelable 对象封送时，它就持有了该对象的引用。它将调用对象的 `writeToParcel` 方法传递一个 `Parcel`。这个对象负责将其状态封送到传递的 Parcel 中。

当框架需要从 Parcel 解除封送时，就从 Parcel 中读取标记（tokens）。Parcel 中的一个标记指出下一个解除封送的对象类型。框架可以利用这个标记识别要解除封送对象的 Java 类。由于 `CREATOR` 是该类的公有静态成员，因此 unmarshaler 很容易就可以为要解除封送的对象获取对 `Parcelable.Creator` 的引用。框架调用创建者的 `createFromParcel` 方法传递 Parcel。创建者负责从 Parcel 中读取之前封送的状态，并返回一个新对象。

Parcelable 中的 Android 文档是明确且完整的。并且，其中有一些有用的技巧。

首先，`createFromParcel` 的常见实现是对一个构造函数的调用。为了让封送和未封送的对象彼此靠近，很多开发人员使用一种特殊包保护的构造函数，这个构造函数使用 Parcel 作为唯一的参数。还有一些开发人员喜欢在 `createFromParcel` 方法中读取 Parcel，然后调用完全指定对象状态的构造函数。

非卖品，仅供非商业用途或交流学习使用

接着，开发人员可能会惊讶地发现，状态从 Parcel 中解除封送的顺序与它被封送的顺序完全相同。如果 writeToParcel 写下一个 String 和一个 int，createFromParcel 就会读取一个 String 和一个 int。

最后，虽然 Parcelable API 会对 Parcelable 进行快速封送和解除封送（大约是 Android 平台上 Java 序列化的两倍），但它使它们几乎不可能子类化。构建一个跨进程 API 可以避免很多麻烦，这个 API 最小化了 Parcelable 继承的需求。

6.5.2 Messenger

Android 的 Messenger 类已经出现两次了。Messenger 是 Message 类（见 5.2 节）远程化的原因。它们还是本章前面提到的，从 IntentService 返回结果的方式。

从本质上讲，Messenger 是对 Handler 中 Parcelable 的引用。小于 Looper 的工作者线程 Handler 能够使能一个外部线程的入队消息。Messenger 延伸了这个能力，使代码对 Handler 的引用传递到它可以传递 Parcelable 的任何地方。尤其是，在 Intent 中传递 Messenger 可以将不同进程中的 Intent 消息添加到队列中进行局部处理。清单 6.15 说明了这一点。

清单 6.15　使用 Messenger

```
public class MessengerServiceHelper {
  public interface MessengerCallbacks { void onComplete(String val); }

  public static final String SERVICE_PACKAGE
    = "net.callmeike.android.messengersandbox";
  public static final String ACTION_SEND
    = "net.callmeike.android.messengersandbox.svc.action.SEND";
  public static final String EXTRA_PARAM_BASE
    = "net.callmeike.android.messengersandbox.svc.extra.BASE";
  public static final String EXTRA_PARAM_MESSENGER
    = "net.callmeike.android.messengersandbox.svc.extra.MESSENGER";
  public static final String EXTRA_RESPONSE
    = "net.callmeike.android.messengersandbox.svc.extra.RESPONSE";

  public static final int WHAT_REPLY = -1;

  public static void startActionSend(
    Context context,
    String base,
```

非卖品，仅供非商业用途或交流学习使用

```
        final MessengerCallbacks callbacks)
    {
      Messenger messenger = new Messenger(new Handler(Looper.myLooper()) {
        @Override
        public void handleMessage(Message msg) {
          switch (msg.what) {
            case WHAT_REPLY:
              Bundle resp = (Bundle) msg.obj;
              callbacks.onComplete(resp.getString(EXTRA_RESPONSE, ""));
              break;
            default:
              super.handleMessage(msg);
          }
        }
      });

      Intent intent = new Intent();
      intent.setPackage(SERVICE_PACKAGE);
      intent.setAction(ACTION_SEND);
      intent.putExtra(EXTRA_PARAM_BASE, base);
      intent.putExtra(EXTRA_PARAM_MESSENGER, messenger);
      context.startService(intent);
    }
  }
```

考虑在一个应用程序中想要使用 MessengerService 的客户端，它是完全独立的应用程序 net.callmeike.android.messengersandbox 的一部分。它通过创建一个 MessengerServiceHelper 实例并调用其方法 startActionSend 可以做到这一点。

魔法在 startActionSend 中。它为线程创建一个封装 Handler 的 Messenger，在线程中会调用方法（Looper.myLooper），将它添加到 Intent 中，然后启动 Intent。

MessengerService 如清单 6.16 所示。它接收辅助方法发送的 Intent 作为调用 onHandleIntent 方法的参数，检索辅助方法包含两个参数，然后用它们调用 handleActionSend。

非卖品，仅供非商业用途或交流学习使用

清单 6.16　Messenger Service

```
public class MessengerService extends IntentService {
  private static final String TAG = "MessengerSvc";

  public MessengerService() {
    super(TAG);
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    if (intent == null) { return; }

  final String action = intent.getAction();
  switch (action) {
    case MessengerServiceHelper.ACTION_SEND:
      handleActionSend(
        intent.getStringExtra(
          MessengerServiceHelper.EXTRA_PARAM_BASE),
        (Messenger) intent.getParcelableExtra(
          MessengerServiceHelper.EXTRA_PARAM_MESSENGER));
      break;
    default:
      Log.i(TAG, "unexpected action: " + action);
    }
  }

  private void handleActionSend(String base, Messenger messenger) {
    Bundle resp = new Bundle();
    resp.putString(MessengerServiceHelper.EXTRA_RESPONSE, base + "
roundtrip!");
    try {
      messenger.send(
        Message.obtain(null, MessengerServiceHelper.WHAT_REPLY, resp));
    }
    catch (RemoteException e) {
      Log.e(TAG, "reply failed: ", e);
    }
  }
}
```

非卖品，仅供非商业用途或交流学习使用

handleActionSend 执行它的任务（在这个清单中，是将“roundtrip”连接到所传递字符串的简单任务），然后准备好返回结果。返回结果根据第 5 章内容可知：从池中获取消息的代码，然后使用所传递的 Messenger 将它添加到队列中，以便 Looper 处理。

这里有一个值得注意的细节。和第 5 章的 Handler 不同，这个 Messenger 准备在进程之间发送 Message。它必须可以对发送的对象封送和解除封送。因此，如果消息字段 obj 包含对一个对象的引用，那么这个对象必须是 Parcelable。事实上，Messenger 方法 send 通过检查 obj 所指向对象的类型可以验证它。

这个清单的返回值是 String。虽然 String 是一个原始可打包的类型，但它没有实现 Parcelable。尝试用 Messenger 发送 String 会产生如下所示的错误消息：

```
E/AndroidRuntime: FATAL EXCEPTION: IntentService[MessengerSvc]
Process: net.callmeike.android.messengersandbox:service, PID: 2106
java.lang.RuntimeException:
    Can't marshal non-Parcelable objects across processes.
```

然而，正如清单所示，避免这种错误很简单。Android 的 Bundle 类型会扩展 Parcelable，并且它知道如何封装 String。将返回值封装到 Bundle 就可以解决这个问题了。

返回清单 6.15，从 Service 发送的消息会出队回到客户端进程中，并且作为调用 Handler 的 handleMessage 方法的参数。该方法解析消息，并用这个结果调用之前注册的回调处理函数。

虽然这段代码很漂亮，但不要让它导致你忘记了生命周期的问题。例如，如果传递的回调处理函数是对 Activity 的引用，那么在它引用的 Handler 被收集前，不能释放这个 Activity 进行垃圾收集。只有在引用它的消息从 Looper 队列中删除后才可以发生这种情况。当然，如果在这之前销毁 Activity，那么这个 Activity 就可能处于不一致的状态，并且对 onComplete 的调用可能会激增。

6.5.3 使用 AIDL

AIDL（Android 接口定义语言）可能是从简单的 Parcelable 开始的下一步。它支持类 Java 方法调用，并通过 Binder IPC 机制返回值。AIDL 是一种多功能语言，对它的完整讨论超出了本书的范围。在 Android 文档中有对它的详细讨论。以下是它的简

要介绍。

AIDL 编译器是 IPC 机制中某种常见工具的实例。它的工作是在给定的接口规范下，产生实现代理和存根的 Java 代码。AIDL 语言是 AIDL 编译器的源语言，看起来很像 Java。

使用 AIDL 与由在另一个进程中运行的由 Service 管理的对象通信是一个多步过程。

1）**定义接口**：清单 6.17 显示了一个简单的 AIDL 定义。要注意的是，Service 的客户端（带有 AIDL 定义的 API）需要这个 API 的 Java 类。它们可以通过编译共享 AIDL，编译共享 Java 代码或者通过使用包含已编译 API 类型的库获取它们。带有公有接口的 Service 必须以这三种方式中的一种发布其 API。将 API（也许还有约定）放入一个独立可共享的模块中很有意义。

清单 6.17　一个简单的 AIDL 定义

```
package net.callmeike.android.servicesandbox.svc;

interface ILogger {
  void startLogging();
  void stopLogging();
}
```

2）**编译接口**：检查编译的输出是有指导意义的。Android 标准建构工具 Gradle 将编译的 AIDL 输出放在<builddirectory>/generated/source/aidl/<variant>/<classpath>/<interface>目录中。例如，清单 6.17 中为应用程序的调试版本构建的 ILogger AIDL 定义在文件…/generated/source/aidl/debug/net/callmeike/Servicesandbox/Service/svc/ILogger.java 中。

清单 6.18 显示了 AIDL 输出的注释摘录。令人惊讶的也许是，一个 Java 源文件是代理和存根的实现。

清单 6.18　AIDL 产生的代码

```
package net.callmeike.android.servicesandbox.svc;

public interface ILogger extends android.os.IInterface
{
  // The service side extends this Stub
```

```
public static abstract class Stub
  extends android.os.Binder
  implements net.callmeike.android.servicesandbox.svc.ILogger
{

// Code elided ...

@Override public boolean onTransact(
  int code,
  android.os.Parcel data,
  android.os.Parcel reply,
  int flags)
  throws android.os.RemoteException
{
  switch (code)
  {
    // when this opcode appears in the stream,
    // call the implementation of startLogging
    case TRANSACTION_startLogging:
    {
      data.enforceInterface(DESCRIPTOR);
      this.startLogging();
      reply.writeNoException();
      return true;
    }

    // Code elided ...

  }
  return super.onTransact(code, data, reply, flags);
}

// The client makes calls to this proxy
private static class Proxy
  implements net.callmeike.android.servicesandbox.svc.ILogger
{
  // the proxy has a stream to the remote
```

```
    private android.os.IBinder mRemote;
    Proxy(android.os.IBinder remote)
    {
      mRemote = remote;
    }

    // ... code elided

    // a call to the startLogging method
    // marshals the arguments into the stream
    @Override public void startLogging() throws android.os.RemoteException
    {
      android.os.Parcel _data = android.os.Parcel.obtain();
      android.os.Parcel _reply = android.os.Parcel.obtain();
      try {
        _data.writeInterfaceToken(DESCRIPTOR);
        mRemote.transact(Stub.TRANSACTION_startLogging, _data, _reply, 0);
        _reply.readException();
      }
      finally {
        _reply.recycle();
        _data.recycle();
      }
    }

    // ... code elided

    }

    // here are the definitions of the opcodes
    static final int TRANSACTION_startLogging
      = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
    static final int TRANSACTION_stopLogging
      = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
}

// here is the interface that the proxy and the implementation implement
```

```
    public void startLogging() throws android.os.RemoteException;
    public void stopLogging() throws android.os.RemoteException;
}
```

3）**实现接口**：这是任务的实际实现。重要的部分是第一行，类的定义。它必须扩展 AIDL 产生的存根，而存根反过来实现 AIDL 定义的接口并扩展 IPC 机制——Binder。清单 6.19 显示了一个远程对象的实现。

清单 6.19 托管对象实现样例

```
public class Logger extends ILogger.Stub {

  @Override
  public void startLogging() {
    // ...
  }
  public void stopLogging() {
    // ...
  }
}
```

4）**从 Service 返回对象**：本章的好几个清单都展示了从绑定型服务中返回托管对象的进程。Service 必须在配置文件中声明，必须小心创建托管对象的一个实例，并且必须从 `onBind` 调用中返回它。

要注意的是，这次客户端在其 `onServiceConnected` 调用中接收到的对象绝对和 Service 返回的对象不同。相反，它是从 Binder 到远程的连接。

5）**将 Binder 连接封装到代理中**：客户端必须访问 Service 使用的相同的 API 定义。虽然当客户端和服务在同一个进程中时，客户端必须将它从 `onServiceConnected` 调用中接收到的对象进行类型转换，但这次它通过使用 `asInterface` 方法而产生的 AIDL 存根会封装所传递的引用——到远程进程的 Binder 连接。连接现在是完整的。清单 6.20 是清单 6.19 中 Service 的客户端实现。

清单 6.20 使用一个远程托管对象

```
private ILogger service;

public void onServiceConnected(ComponentName name, IBinder binder) {
```

严禁网络传播本PDF，违者责任自负！

```
    service = ILogger.Stub.asInterface(binder);
    button.setEnabled(true)
  }

  void startLogger() {
    service.startLogging();
  }
```

6.5.4　创建进程

在远程进程中运行会有很大的开销。所有的封送、解除封送以及在地址空间中的备份都有开销。然而，这种开销有时是值得的。仅举一个实例，远程任务的崩溃不会影响其客户端。

在一个不同进程中运行任务最明显的方式是在一个不同的应用程序中运行它。这只是使用 Intent 处理的 Service 问题，这个 Service 是不同应用程序的一部分。

如果目标设备已经安装了所需的应用程序，那么它能正常工作。当应用程序必须负责安排多个进程时，若是需要多个应用程序就会很麻烦。例如，只是为了让第一个应用程序正常工作而要求用户安装第二个应用程序，会引发用户的不满。

注意

没有要求应用程序有 UI。创建只包含一个或者多个 Service 的应用程序是完全可能的。这种应用程序不会出现在 Android 桌面上，也不能由用户显式运行。

这种应用程序甚至可以包含 Activity，只要这个 Activity 没有与 `android.intent.action.MAIN` 动作和 `android.intent.category.LAUNCHER` 类别相匹配的过滤器即可。

创建进程的第二种方法是在应用程序配置文件中使用 `android:process` 组件属性。这个属性指定组件应该在一个进程中运行，而不是在应用程序的默认进程中运行。清单 6.21 是在进程“net.callmeike.android.Servicesandbox:other”上运行的 Service 清单。

清单 6.21　指定组件的进程

```
<service
  android:name=".svc.SimpleService"
```

严禁网络传播本PDF，违者责任自负！

```
    android:exported="true"
    android:process=":other"
    >
    <intent-filter>
      <action android:name="net.callmeike.android.servicesandbox.LOGGER"/>
    </intent-filter>
    <intent-filter>
      <action android:name="net.callmeike.android.servicesandbox.BLOGGER"/>
    </intent-filter>
  </service>
```

Android 区分两种进程类型：本地和远程。**本地进程**是以“:”开头的进程。带有其他包名的其他应用程序不能复制这个名称。如果其他应用程序无法命名进程，那么就不能在它们的进程规范中使用它，也不能将它们的组件放入其中。

此外，**远程进程**有其他进程可以访问的名称。远程名称以一个小写字母开头，并且必须包含“.”字符。如果两个不同的应用程序使用相同的证书签名并且用同一个 Linux 用户 ID 运行，并且这个 ID 给组件指定同一个进程，那么这些组件就会在同一个进程上运行。

清单 6.22 和清单 6.21 几乎相同，但它将进程中的 SimpleService 放入另一个可以放入组件的应用程序中。

清单 6.22 指定组件的进程

```
  <service
    android:name=".svc.SimpleService"
    android:exported="true"
    android:process=" com.callmeike.android.shared"
    >
    <intent-filter>
      <action android:name="net.callmeike.android.servicesandbox.LOGGER"/>
    </intent-filter>
    <intent-filter>
      <action android:name="net.callmeike.android.servicesandbox.BLOGGER"/>
    </intent-filter>
  </service>
```

严禁网络传播本PDF，违者责任自负！

6.6　浅谈 Binder

如果没有考虑 Binder，那么对 Android 进程间通信的讨论就是不完整的。从真实意义上说，Binder 是 Android 的核心。它支持 Android 的安全机制，独特的跨应用程序组件结构和调用其他应用程序的强大机制，就好像它们是函数一样。

Android 的 Binder 是第 4 代。原来的 Binder 是为 BeOS 开发的。它最开始出现在 Palm 上，然后才发布为 OpenBinder。Android 版本是重实现。每个 Android 开发人员都应该知道 Binder，并对其行为有基本的了解。

6.6.1　Binder 线程

Android 远程进程只是一个不同的进程。所有正常的 Android 调度和线程行为都适用于它。尤其是，在远程进程中调用 Service 生命周期方法，就好像它们总是从进程主线程上调用一样。然而，远程线程中 Service 托管对象的方法是从 Binder 线程上调用的。

Binder 线程是为 Binder 使用而显式创建的线程。当来自一个或者多个远程客户端进程的请求排队等待绑定型服务的执行时，内核就要求服务器进程的框架启动新线程来处理它们。这些新线程一旦启动，就要求在内核中执行工作。

应用程序不会启动超过 15 个的线程。Android 中编译一种软件限制。虽然这个数字可能会随着时间的推移或者设备供应商的不同而改变，但主流 Android 代码在 Android 存在的十几年间都没有改变这个限制。

6.6.2　Binder 数据传输限制

Binder 用于传输数据的内存缓冲区在大小上是受限制的。再次声明，虽然细节会随着时间的推移和供应商的不同而不同，但基本原则是，不可能在一次传输中移动超过 1MB 的数据。

但还是有办法的。由于 Binder 是内核的一部分，因此它可以引用内核私有结构。这种结构是打开的文件。这样，父进程和子进程都可以在 `fork` 调用后引用同一个打开的文件，因此一个进程可以打开一个文件，并通过 Binder 将它传递给另一个进程。将大图像传递给 Service 的客户端可以将图像作为文件来打开，并将打开的文件传递

严禁网络传播本PDF，违者责任自负！

给 Service，而不是通过 Binder 传递图像的每个字节。

6.6.3 绑定到死

Service 发现客户端已经终止，有时这是很有必要的。当 Service 管理稀缺或者昂贵的资源，并且不能容忍客户端请求但不释放它们时，这一点尤为重要。

当Service及其客户端之间的连接为Service提供一种调用客户端的方式时，Service就会持有这个客户端调用的 `IBinder`。`IBinder` 接口定义了方法 `linkToDeath`。通过调用该方法，Service 会在客户端连接失败时注册一个回调。

这是可能的，因为 Binder 是内核的一部分。内核不监视进程的状态；它是创建进程的事情。不需要心跳或者轮询。内核保证知道客户端进程是否失败。Binder 将信息传递给已注册的回调。

6.7 小结

Service 对开发人员来说是重要的工具，虽然从架构上来说它们是不能存在于其他地方的某些功能。在 Android 环境中，当有价值的工作在 UI 中不可见时，它们是调整进程优先级唯一的一种方式。对于在单独线程上运行，甚至在单独进程中运行的任务，启动型服务和绑定型服务分别提供了更窄的接口和更宽的接口。

严禁网络传播本PDF，违者责任自负！

第 7 章
定时任务

我们发现，通信的大敌是对它的错觉。

威廉·怀特

最重要的一种异步任务是必须在调度中重复的任务。定期任务通常会用一些外部状态同步包含这些任务的应用程序状态。当然，外部状态将更改通知推送给应用程序就再好不过了——没有粒度延迟和尺度的推送通知要比轮询好得多。

然而，有时需要轮询和显式调度。不需要发送推送通知的资源依赖于一个应用程序，该程序必须轮询该资源。或者，设备的本地条件驱动显式调用。当所需要的条件都有效时，需要执行操作的应用程序可能要再次调用该操作，该操作应该只发生在设备被连接到电源或者按流量计费的网络中时。

7.1 任务特性

“重复”或者“调度”任务的一般范畴过于宽泛。它包含像每 17ms 刷新一次屏幕、每天给设备充电等事情。这些事情都很有趣，对于构建响应性强、资源节约型的应用程序来说很重要。然而，本章关注一组更具体的任务。

这里考虑的任务是一个可以多次执行的代码路径。它们有一段介于一分钟和一天之间的周期。应用程序将它作为业务逻辑的一部分来调度，并将其自身与环境同步。一些清单在追踪跑步者的路径时可能会记录路标，将新照片推送给远程服务器，或者轮询第三方网站进行更新。当安装新媒体或者在一个长时间快速移动的列表视图中从远程源加载图像来显示时，不包括要执行的任务。

这个受限列表页甚至会包含很难做好的事情。组件生命周期、异步通知和跨线程

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

共享可变数据等常见问题都适用。帮助隐藏复杂性并且让它更容易做正确事情的框架是非常好的。

这种框架会是什么样呢？完美的异步调度框架会为开发人员做什么呢？愿望清单看起来是下面所示的样子：

- 框架应该支持线程安全；
- 框架应该是 Android 组件生命周期感知型的；
- 框架应该巧妙使用进程优先级；
- 框架应该是电量节约型的；

让我们详细检查这些需求。

7.1.1 线程安全

异步调度机制可能涉及多个线程。如果是这样，前一章所讨论的并发代码固有的所有复杂性在这里都适用。一个好的框架应该很容易知道哪个代码在哪个线程中运行，从而防止意外的共享状态。它还应该提供一种安全发布机制，以便在需要状态共享时使用。

7.1.2 生命周期感知

一个好的调度框架应该能正确地与 Android 组件生命周期进行交互。即使框架持有对已销毁组件的引用，也不会导致内存泄漏。

理想情况下，即使调度该事件的应用程序不在运行中，也可以调度该事件。预定事件应该重启应用程序。

7.1.3 巧妙使用进程优先级

当调度这些事件的应用程序在屏幕上不可见，没有运行，甚至是在手机处于低电量模式时，支持调度事件的调度框架必须准备好管理进程优先级和**唤醒锁**。如果设备处于低电量模式，那么框架必须捕获一个唤醒锁，防止它在事件完全处理完之前恢复低电量模式。

同样重要的是，一旦事件被处理，它就必须释放唤醒锁来尽量减少电池的影响。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

一个好框架还必须调整驱动它的进程优先级（减小 oom_adj），通知 Android 的进程管理系统，进程正在做有用的工作，不应该为释放内存而终止。

注意

唤醒锁是 Android 使用的一种低电量管理功能，以便在设备进行有用的工作时保持设备有电。Android 使用一种名为“opportunistic suspension”的功能管理电源。当 opportunistic suspension 启动时，内核就会非常积极地进入低电量状态。它让系统休眠，除非至少有一个唤醒锁。用户空间应用程序负责在必要时捕获并释放唤醒锁。

唤醒锁最初由 Android 开发团队加入到 Linux 内核的 Android 分支中。经过几年激烈的辩论后，现在主线 Linux 内核有一个支持唤醒锁功能的“suspend blockers”。

当设备处于低电量模式时，肯定不存在唤醒锁。如果事件在设备处于这种状态时发生，那么事件接收方必须确保尽快捕获唤醒锁，否则设备会在事件完全处理前返回休眠状态。

7.1.4　电量节约

节约电量的方式有很多种，但最重要的一种是将任务集群。

假设有一台设备，其所有者已经安装了几个应用程序，而每个应用程序每天需要进行二三十分钟的同步。这些应用程序的设计者尽全力在对用户影响最小的晚上来调度这项工作，在午夜到凌晨 5:00 随机选择半小时。随着这类应用程序数量的增加，设备在晚上消耗大量电量的可能性也随之增加。如果所有应用程序同时在某半小时内同步，对电池的影响可以大幅度降低，这样设备在晚上剩余的时间都可以休眠。

当任务使用蜂窝无线电时，任务集群变得更加重要。由于给蜂窝无线电供电通常是设备做的第二昂贵的事，因此集群网络的使用是解决电量的一种重要方式。

用户应用程序不可能看到其他应用程序定期调度任务的全貌。集群任务是只能在系统范围内完成的任务。

注意

蜂窝无线电电源管理是一个复杂且有趣的主题。大多数现代无线电都有 3 种状态：低电量状态、待机状态和活动状态。了解这些状态和如何使用它们可能意味着需要知道电量节约型应用程序和电池直方图（battery hog）的不同。Ilya Grigorik 的《High Performance Browser Networking》（O’Reilly，2013）包含了对这个问题清晰完整的讨论，对设计连接移动应用程序的任何人来说，这都是宝贵的资源。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

7.1.5 记分卡

表 7.1 显示了作为记分卡的需求列表，用于评估各种调度技术。每个框架在 4 个轴上各占 0～5 分。

表 7.1 记分卡

框 架	线 程 安 全	生命周期感知	进程优先级感知	电 量 节 约
一个框架	0	2	4	5

接下来评估一些框架。

7.2 Timer 和 TimerTask

为了保证完整性，我们在这里包含了 Java 的 `Timer` 和 `TimerTask` 类。一些 Android 应用程序仍然在使用它们，并且堆栈溢出的答案偶尔也会推荐它们。然而即使在桌面 Java 中，它们也是要被淘汰的。现在的应用程序使用 Executor 框架和 `ScheduledThreadPoolExecutor`。

第 2 章讨论了避免任意线程创建的原因。这个讨论直接适用于 `Timer`。应用程序应该对它产生的线程数有相应的策略。为新任务创建新线程（像 `TimerTask` 那样）不是一种好策略。

表 7.2 的记分板包含了 Timer，但不会对它记分。不要在你的（Android 或者其他）应用程序中使用 `Timer` 或 `TimerTask`。

表 7.2 记分卡：定时器任务

框 架	线 程 安 全	生命周期感知	进程优先级感知	电 量 节 约
Timer 和 TimerTask	NA	NA	NA	NA

7.3 Looper/Handler

第 5 章详细讨论了 Looper/Handler 框架。讨论表明了在将来的某个时间会调度任务执行。只需要在每次运行时重新安排任务即可将它转换为定期调度的框架。清单 7.1 是一个基于 Looper/Handler 的简单的定期调度框架的清单。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

清单 7.1　定期 Looper 任务

```
public class PeriodicTaskScheduler {
  private static final int PERIODIC_TASK = -1230;

  public interface Task { void onTask(); }

  public static final class TaskInfo {
    final long interval;
    final Task task;
    volatile boolean cancelled;
    TaskInfo(Task task, long interval) {
      this.task = task;
      this.interval = interval;
    }
  }

  private static class PeriodicTaskHandler extends Handler {
    public PeriodicTaskHandler(Looper looper) { super(looper); }

    @Override
    public void handleMessage(Message msg) {
      if (msg.what != PERIODIC_TASK) { return; }
      long t = SystemClock.elapsedRealtime();

      TaskInfo info = (TaskInfo) msg.obj;
      if (!info.cancelled) {
        info.task.onTask();
        scheduleTask(msg.what, info, t);
      }
    }

    public void scheduleTask(int what, TaskInfo info, long t) {
      Message msg = obtainMessage(what, info);
      sendMessageAtTime(msg, t + info.interval);
    }
  }
```

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
    private final PeriodicTaskHandler handler;

    public PeriodicTaskScheduler(Looper looper) {
      handler = new PeriodicTaskHandler(looper);
    }

    public TaskInfo scheduleTask(Task task, long interval) {
      TaskInfo info = new TaskInfo(task, interval);
      handler.scheduleTask(PERIODIC_TASK, info, SystemClock.elapsedRealtime());
      return info;
    }

    public void stopTask(TaskInfo info) {
      info.cancelled = true;
    }
  }
```

这种调度框架有很多值得推荐的地方。它功能齐全，可以使用现有的 Looper，也可以使用为此专门创建的 Looper。任务可以单独调度和取消。代码容量很小，可以节约资源。如第 5 章所述，它对垃圾回收的影响很小。

然而，它有一些很大的缺点。最明显的两个缺点是相同的，它们在 Android 的很多并发结构中都存在：线程安全和生命周期不一致。`onEvent` 方法的回调会发生在工作者线程与初始化调度程序 Looper 相关联的地方。实现 `Task` 接口的类很可能必须是线程安全的。

效仿 AsyncTask 的实现，可以扩展框架来将 `onEvent` 调用入队到原来调用 `scheduleTask` 的 Looper 中。作为 `scheduleTask` 参数对于传递的 Looper，它甚至可以将其入队。

然而，这些改变都没有解决生命周期的问题。定期执行框架持有对 Task 的引用，它定期执行入队，直到这个任务停止为止。如果任务直接或者通过其他对象持有对 Android 组件的引用，它就会遭受与 AsyncTask 相同的问题，如第 4 章所述。

使用与清单 7.1 类似的框架时还有其他问题。由于 Looper/Handler 框架根据调度时间将一个新条目排序到其任务队列中，因此，比调度间隔运行时间更长的任务可能会饿死同一个队列中的其他任务。此外还要注意，这个框架调度的任务非常短暂；它们既不会阻止 Android 终止一个应用程序，也不会饿死这种终止。

版权所有，严禁传播，违者自负法律责任！

表 7.3 对记分卡中的 Looper/Handler 框架记分。该评级过于悲观。和 AsyncTasks 一样，Looper/Handler 定期调度框架可以很好地处理某些种类的任务。虽然完全不适合网络交互，但它们非常适用于使用当前时间的更新显示（选择一个明显的清单）。

表 7.3　记分卡：Looper/Handler

框　架	线 程 安 全	生命周期感知	进程优先级感知	电 量 节 约
Timer 和 TimerTask	NA	NA	NA	NA
Looper/Handler	1	0	0	1

7.4　基于自定义服务的调度程序

很多开发人员最终都会设计一个自定义异步执行服务。通常，这些服务基于 `ThreadPoolExecutor` 和一些为执行提交任务所代表的对象。这种任务可以被扩展以支持定期任务。这种扩展可以使用 `ScheduledThreadPoolExecutor`，并将任务对象扩展从而包含调度信息。

然而，这种服务的设计细节取决于应用程序的需求。例如，服务可能非常简单，需要重新调度自己的任务（就像它们在 Looper/Handler 清单中所做的那样）。此外，服务可以支持复杂的调度选项，这些选项可以指定调度特定事件所需的连接类型、电源状态，甚至是物理位置。尝试为自定义调度程序的所有（或者多个）可能的实现提供代码超出了本书的范围。我们的讨论必须基于共性。

再次声明，通用方法有很多优点。可以定制自定义时间表（或者甚至扩展到应用程序的整个生命周期），以具有应用程序需要的功能。一个结构良好的服务可以调整进程优先级，这样就不能在做有意义的工作时中断应用程序了。它还会努力将线程安全问题隔离开，从而避免意外错误的发生。

然而，选择创建并维护自定义调度程序是一个重大的决定。所有缺陷都属于作为开发人员的你，并且实现只在你和你的团队改进它时才会得到提高。此外，作为应用程序一部分的调度服务对电源和连接等系统级信息的访问是非常有限的。它永远不会有任务集群所必需的系统范围视图。

表 7.4 所示为基于自定义服务的调度程序，它能很好地运行。很多最常见最流行的应用程序使用其中的一个或者多个。然而，它们的代码太多了，如果有更好的选择就太好了。

版权所有，严禁传播，违者自负法律责任！

表 7.4 记分卡：基于自定义服务的调度程序

框　架	线程安全	生命周期感知	进程优先级感知	电量节约
Time 和 TimerTask	NA	NA	NA	NA
Looper/Handler	1	0	0	1
基于服务的调度程序	3	2	3	2

7.5 Alarm Manager 和 Intent Service

第 6 章引入了 IntentService，它是 service 和 Looper/Handler 的简单组合，它是由 Intent 识别的并在后台线程上执行的任务。将 IntentService 和 AlarmManager 组合起来可以创建一个非常有用并且可以调度定时任务的框架。

AlarmManager 是一种系统级服务，可以在未来的某个时刻调度 PendingIntent 的交付。它能够唤醒处于低电量状态的设备，并调度在进程边界上持久存在的警报。清单 7.2 显示的是使用 AlarmManager 启动和停止一个样例任务的执行。

清单 7.2 用 AlarmManager 进行定时调度

```
public class AlarmScheduler extends IntentService {
  private static final String TAG = "SCHEDULER";

  private static final String PARAM_TASK = TAG + ".TASK";
  private static final int SAMPLE_TASK = -1;

  public static void startSampleTask(Context ctxt) {
    int interval
      = ctxt.getResources().getInteger(R.integer.sample_task_interval);
    ((AlarmManager) ctxt.getSystemService(Context.ALARM_SERVICE))
      .setInexactRepeating(
        AlarmManager.ELAPSED_REALTIME,
        SystemClock.elapsedRealtime() + 100,
        TimeUnit.MILLISECONDS.convert(interval, TimeUnit.MINUTES),
        getTaskIntent(ctxt, SAMPLE_TASK));
  }

  public static void stopSampleTask(Context ctxt) {
    ((AlarmManager) ctxt.getSystemService(Context.ALARM_SERVICE))
```

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
        .cancel(getTaskIntent(ctxt, SAMPLE_TASK));
    }

    private static PendingIntent getTaskIntent(Context ctxt, int taskId) {
      Intent intent = new Intent(ctxt, AlarmScheduler.class);
      intent.putExtra(PARAM_TASK, taskId);
      return PendingIntent.getService(
        ctxt,
        taskId,
        intent,
        PendingIntent.FLAG_UPDATE_CURRENT);
    }

    // ...
}
```

注意

PendingIntent 是一种让外部进程执行操作的方式，就好像它是最初为这个操作创建的进程一样。

可以将 PendingIntent 看作给未来的笔记。一个应用程序可以将指令放在 PendingIntent 信封中，然后将这个信封递交给其他应用程序。在某个时刻，其他应用程序将这个信封放在邮件中。原始应用程序接收到信封，打开它，并执行之前编写的指令。

在清单 7.2 中，静态方法 `startSampleTask` 获取 `AlarmManager` 的引用，并通过调用方法 `setInexactRepeating` 调度 `PendingIntent` 进行定期交付。`SetInexact Repeating` 的第一个参数是调度模式。第二个和第三个参数控制 PendingIntent 第一次交付的时间，以及之后的重复间隔。稍后我们会详细讨论所有这些参数。

静态方法 `stopSampleTask` 取消定期任务。它通过使用与 `startSampleTask` 相同的方法创建 PendingIntent：`getTaskIntent`。然后，它调用 `AlarmManager` 方法 `Cancel`。使用同样的方法为启动和停止方法产生 PendingIntent，从而保证它们是相同的。尤其是，在 `getTaskIntent` 中 `getService` 方法的第二个参数的整数标志在这两个方法中是相同的。带有不同标志的 PendingIntent 不可能相同。

`getService` 第四个参数的标志 `FLAG_UPDATE_CURRENT`，允许多次刷新并交付

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

PendingIntent。还有很多其他标志。不要在重复任务中使用 ONE_SHOT。

驱动样例任务执行的 Intent 是在 getTaskIntent 前几行中创建的。它是交付给 AlarmScheduler onHandleIntent 方法的显式 Intent。正如在 Looper/Handler 中一样，操作码（在带有关键字 PARAM_TASK 的 Intent 中保存的值）会识别要执行的具体任务 SAMPLE_TASK。

7.5.1 AlarmManagerService

实际上是系统服务 AlarmManagerService 传递调度的 PendingIntent。AlarmManagerervice 是独立应用程序中的组件，可以在设备处于低电量状态时与硬件交互来唤醒并交付事件。Android 系统在启动时运行它，并在设备没电时保持运行。它总是在运行。这是任何用户应用程序都不能做的事情。因此，它可以重启已经停止的应用程序，将调度的 Intent 交付给它。

AlarmManager API 是 AlarmManagerService 进程间通信连接的瘦封装器。对本地 AlarmManager 的调用会不加修改地发送给 AlarmManagerService。

AlarmManagerService 会拒绝调度过去的事件。由于调用 setInexactRepeating 实际上是 IPC，因此它会消耗大量的时间。尝试在当前时间调度一个事件，有时会失败，因为当服务器处理消息时，原来的当前时间现在已经过去了。一种好的做法是在不远的未来调度第一个事件：100ms 就足够了。

由 AlarmManagerService 支持的调度模式可能会引起混乱。当这些模式的应用程序在 Android 的演进过程中发生改变时，尤为如此。忽略一次执行的任务（为一次执行调度的任务），任务调度中有 3 个维度：时钟类型、调度窗口大小和电源方案。

时钟类型是最简单的维度。有两种类型的时钟：挂钟 RTC 和计时时钟 ELAPSED_REALTIME。前者使用 System.currentTimeMillis，后者使用 SystemClock.elapsedRealtime。这两种设备都在设备休眠期间计算时间。但它们的不同点在于，挂钟可以倒退（由用户或者时钟网络设置），但计时时钟不可以。使用 RTC 在每天 18:00 调度事件。使用 ELAPSED_REALTIME 每 24 小时调度一个事件。

一致性是必不可少的。例如，使用 ELAPSED_REALTIME 模式时，一定要使用 SystemClock.elapsedRealtime 调度第一次交付。混淆这两种时钟类型会导致在很远的未来调度才第一次交付，或者根本不会调度。

与时钟类型相关的是调度窗口大小。任务批处理，尤其是那些可能使用无线电的

非卖品，仅供非商业用途或交流学习使用

任务，是记分卡的目标之一。AlarmManagerService 一定会支持任务批处理，但支持会随着释放而改变。

在 API level 19（KitKat）之前，setInexactRepeating 和 setRepeating 方法分别调用进行被批处理和不会进行批处理的任务。不能准确调度的任务可以向上移动到其调度间隔 75%的地方，然后将它们与其他任务组合起来。对于 API level 18 及其之前版本的应用程序也适用。

然而，对于面向 API level 19 及更高的版本，这两种方法是相同的：都是用于不能准确调度的任务。最近的应用程序无法准确调度一个重复的任务。唯一的方法是调度一个一次性任务但要求它在运行时能再次调度自己，就像 Looper/Handler 框架那样。

使用 IPC 的 AlarmManagerService、PendingIntent 和硬件接口，这是一种用于调度频繁任务的相当重量级的机制。开销和批处理导致的时间变化，使得以几分钟间隔重复的定时任务变得不明智。最新版本的 Android 甚至不可能在小于 1min 的时间间隔内调度任务。

虽然 AlarmManagerService 可以将设备从低电量状态唤醒，但清单 7.2 中的代码不会这么做，因为它没有使用 _WAKEUP 调度模式。在非唤醒模式中，AlarmManagerService 将关键字 Intent.EXTRA_ALARM_COUNT 添加到 Intent 的 Bundle 附加设备中，它将交付给目标应用程序。与这个关键字相关的值是一个整数，它给出了未交付的事件数，因为设备在事件调度时处于休眠状态。

使用_WAKEUP 调度模式对电池来说是很昂贵的，这并不是一个可以轻易做出的决定。它还要求对代码进行一些更改。除非有人捕获唤醒锁，内核的“opportunistic suspension”机制才会强迫设备立即回到低电量状态。AlarmManagerService 在处理警报的过程中持有唤醒锁。然而，一旦它交付，PendingIntent 就会释放唤醒锁。奇怪的是，“交付”的含义取决于 PendingIntent 的目标是 BoardcastReceiver 还是 Service。所有 PendingIntent 都不是平等创建的。

在清单 7.2 中，PendingIntent 是使用 getService 方法调用创建的。当 AlarmManagerService 交付了这种 PendingIntent 时，它会在目标应用程序确认接收 Intent 前释放唤醒锁。在这种情况下无法保证不存在唤醒锁缺口，AlarmManagerService 可以在目标应用程序捕获其唤醒锁前释放它的唤醒锁。

幸运的是，用 getBoardcast 方法创建的 PendingIntent 的处理方法有所不同。在目标应用程序的 onReceive 方法返回前，AlarmManagerService 不会释放其唤醒锁。捕获 onReceive 方法中的唤醒锁可以保证没有缺口。为了安全使用_WAKEUP 调度模式，

非卖品，仅供非商业用途或交流学习使用

PendingIntent 必须被交付给在 onReceive 方法中捕获唤醒锁的 BroadcastReceiver。Android 提供的正是这种 BroadcastReceiver——WakefulBroadcastReceiver。清单 7.3 演示了接收安全唤醒系统事件所必须进行的更改。

清单 7.3 用报警管理器进行低电量定期调度

```
// ...

public static class AlarmReceiver extends WakefulBroadcastReceiver {
  @Override
  public void onReceive(Context ctxt, Intent intent) {
    Intent svcIntent = new Intent(ctxt, AlarmScheduler.class);
    svcIntent.putExtras(intent);
    startWakefulService(ctxt, svcIntent);
  }
}

public static void startSampleTask(Context ctxt) {
  int interval = ctxt.getResources().getInteger(R.integer.sample_task_interval);
  ((AlarmManager) ctxt.getSystemService(Context.ALARM_SERVICE))
    .setInexactRepeating(
      AlarmManager.ELAPSED_REALTIME_WAKEUP, //!!! Wakeup!
      SystemClock.elapsedRealtime() + 100,
      TimeUnit.MILLISECONDS.convert(interval, TimeUnit.MINUTES),
      getTaskIntent(ctxt, SAMPLE_TASK));
}

// ...
private static PendingIntent getTaskIntent(Context ctxt, int taskId) {
  Intent intent = new Intent(ctxt, AlarmReceiver.class); // target receiver
  intent.putExtra(PARAM_TASK, taskId);
  return PendingIntent.getBroadcast( // target receiver
    ctxt,
    taskId,
    intent,
    PendingIntent.FLAG_UPDATE_CURRENT);
}
// ...
```

非卖品，仅供非商业用途或交流学习使用

清单 7.3 中最显著的更改是 AlarmReceiver 类的添加。它只是在捕获唤醒锁后，将它从任意 Intent 上接收到的 extras 转发到 AlarmScheduler 中。

AlarmReceiver 是一个 WakefulBroadcastReceiver，是在应用程序配置文件中注册的，如清单 7.4 所示。

清单 7.4　低电量调度程序的配置文件

```
<manifest
  xmlns:android "http://schemas.android.com/apk/res/android"
  package "net.callmeike.android.intentservicescheduler"
  >

  <uses-permission android:name "android.permission.WAKE_LOCK"/>

  <application
    android:allowBackup "false"
    android:icon "@mipmap/ic_launcher"
    android:label "@string/app_name"
    android:supportsRtl "true"
    android:theme "@style/AppTheme"
    >

    <!-- ... -->

    <service android:name ".svc.AlarmScheduler"/>
    <receiver android:name ".svc.AlarmScheduler$AlarmReceiver" />
  </application>

</manifest>
```

清单 7.4 还强调了另一个重要的细节。捕获唤醒锁是一种危险的能力，因此需要得到许可。为了使用 WakefulBroadcastReceiver，应用程序必须请求 WAKE_LOCK 许可。

在清单 7.3 中还有 3 个比较小的更改。首先，在 getTaskIntent 顶端构建的 Intent 指向接收者，而不是服务。其次，封装它的 PendingIntent 是一个广播 Intent (getBroadcast)，而不是服务 Intent(getService)。最后，调用 setInexactRepeating 会用标志 ELAPSED_REALTIME_WAKEUP 唤醒调度模式。

非卖品，仅供非商业用途或交流学习使用

值得注意的是，这种实现并不是完全防弹的。对于释放唤醒锁的应用程序来说，没有绝对安全的方式。考虑到 Android 的进程模式，无法确定释放唤醒锁的代码将被执行，`finally` 阻塞或者以其他方式执行。唯一安全的做法是使用 `acquire(long timeout)` 在限定的时间段内捕获锁，然后在必要时重新捕获它。

7.5.2 可调度的任务

回想一下第 6 章，由 IntentService 执行的任务通常紧紧地绑定到执行它们的服务中。和这里讨论的其他调度程序不同，IntentService 的任务通常不会表示为传递给服务的闭包或观察者。相反，它们通常是执行服务本身代码中的方法。添加一个用于执行的新任务通常意味着将任务编码，并从服务中显式调用它。

这个约束在清单 7.2 和清单 7.3 中非常明显。所有参数（第一个定时事件的时间，后续事件发生的间隔等）都隐藏在调度方法的调用中。例如，创建一个新的可调度任务除了需要任务本身的代码外，还需要两个新方法 `scheduleAnotherTask` 和 `cancelAnotherTask`。当然可以以编程方式放松这些约束，但要安全地放松这些约束并不容易。

首先要注意的是，一个 PendingIntent 只能调度一次。在调度 PendingIntent 之前，`AlarmManagerService` 会为 `equals` 的 PendingIntent 取消所有调度。再次尝试调度 `equals` 的 PendingIntent 会替换这个 intent 之前的调度。

显然，这个问题的解决方案是为每个新调度创建新 PendingIntent。为什么要这么做呢？动态创建的 PendingIntent 可用于表示传递给通用调度程序的任意任务。我们可以扔掉清单 7.2 和清单 7.3 中紧密的任务绑定！这样每个人都能自由地定时调度了。

不幸的是，Android 组件的生命周期是必须遵守的。要使动态创建的 PendingIntent 策略有效，定时 intent 的含义就必须可以在进程边界中以持续的方式保存。

考虑与清单 7.2 所示服务类似的调度程序，它为要调度的新任务动态创建了一个新 PendingIntent。它创建一个新的、唯一的 ID，并使用这个 ID 同时作为 `PARAM_TASK` 和 PendingIntent 的标志。它还使用这个 ID 保存要在 SparseArray 中运行的任务引用。当 Intent 交付时，它在数组中查找任务并运行它。生活真美好！

然而，应用程序最终会被杀死，所有应用程序都是。假设在杀死 `AlarmManagerService` 的 37min 后将设备从深度睡眠中唤醒，重启应用程序，并如实交付 PendingIntent。不幸的是，调度程序的任务数组中没有什么内容！它已经忘记应该做什么了。

非卖品，仅供非商业用途或交流学习使用

一方面，很难让可调度的任务成为可持续的，对于大多数应用程序来说，这只是不合理的努力。另一方面，一些隔离，尤其是让任务的测试独立于调度框架的隔离，似乎是一个好主意。清单 7.5 显示了 AlarmScheduler 的 `onHandleIntent` 方法，它看起来好像使用了依赖注入框架，在这个清单中是 Dagger 2。这种框架将任务从运行它们的 Service 中分离出来。这使得测试任务代码变得更容易。

清单 7.5　报警调度程序任务执行

```
public class AlarmScheduler extends IntentService {

  // ...

  @Inject volatile Lazy<SampleTask> sampleTask;

  public AlarmScheduler() { super(TAG); }

  @Override
  public void onCreate() {
    super.onCreate();
    DaggerTaskComponent.create().inject(this);
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    try {
      int op = intent.getIntExtra(PARAM_TASK, 0);
      switch (op) {
        case SAMPLE_TASK:
          sampleTask.get().run();
          break;
        default:
          Log.w(TAG, "unexpected op: " + op);
      }
    }
    finally {
      AlarmReceiver.completeWakefulIntent(intent);
    }
  }
}
```

非卖品，仅供非商业用途或交流学习使用

注意

虽然依赖注入框架不在本书的范围内，但它是 Android 开发人员的必备工具。它们已经很成熟了，有几种可供选择的依赖注入框架，每一种都有自己的优缺点。它们是实现单元可测试 Android 代码的非常有效的方式。每个 Android 开发人员都至少应该使用一个。

在服务的 `onCreate` 方法中，依赖注入框架在调用 `DaggerTaskCommponent.create().inject()` 期间填充 volatile 变量 `sampleTask`。变量必须是 volatile 的，因为它是从 UI 初始化，然后从工作者线程中使用的。

那么，基于 AlarmManager/IntentService 调度程序的记分卡如何呢？它具有第 6 章讨论的 IntentService 的所有优缺点。它与客户端的隔离使它非常适合于线程和生命周期安全。当定时任务需要在可见视图中修改值时，这种隔离会让人非常沮丧。

IntentService 执行的所有任务都是在一个工作者线程上按顺序执行的。这意味着不会有意外，但长时间运行的任务会迫使后续任务脱离它们的计划。IntentService 具有完美的进程优先级感知。它会在执行任务时启动，并在最后一个任务完成时停止。

只要使用得当，AlarmManager 确实能提供唤醒锁管理。最重要的是，由于它具有系统级范围的权限，因此它可以将任务集群。这是一个非常重要的优点。它是目前所讨论的调度机制中唯一能做到这一点的。

此外，虽然 AlarmManager 在进程中是持久的，但在重启过程中不是持久的。如果有必要，一个应用程序必须捕获 `BOOT_COMPLETE` 广播，在系统启动时重新调度其警报。

此外，这个框架中的任何内容都不支持基于无线电、电源或者网络状态的调度任务。但可以使用没有内置支持的其他 API 来确定一些其他事情。

表 7.5 给出了分数——还不错。虽然这个框架可以很好地满足某些需求，但它不是万能的，还有很多改进的空间。

表 7.5 记分卡：基于自定义服务的调度程序

框　架	线 程 安 全	生命周期感知	进程优先级感知	电 源 节 约
Timer 和 TimerTask	NA	NA	NA	NA
Looper/Handler	1	0	0	1
基于服务的调度程序	3	2	3	2
AlarmManager/IntentService	4	4	4	3

非卖品，仅供非商业用途或交流学习使用

7.6　同步适配器

到目前为止，本章的讨论都是关于实现的，即如何在定期调度上执行任务。也许需要后退一步——也许不应该考虑定期调度任务的方式，而是应该首先考虑所需的定期任务。我们需要把抽象向上移动一层。

所谓的同步适配器就是这样一种更高级的抽象。从本质上讲，它是由“内容”URL 标识的本地数据集和外部信任账户之间的连接。

由同步适配器管理的同步数据集的典型清单是联系人列表。用户的所有设备都有相同的用户联系人视图，因为它们都用同一个中央数据库同步。这个中央数据库由身份验证保护，因此只有所有者可以访问联系人列表。

同步适配器已经存在很长时间了：至少是从 Éclair（API Level 5）开始的。在平台中文档很不完善的早期，即使配置稍微有点错误也会导致整个操作系统崩溃。但最近，它们似乎变得更稳定了，文档也有了显著的改进（尽管到了今天，一个错误实现的账户也会导致系统设置应用程序崩溃）。现在，将同步适配器添加到一个应用程序中已经是一种相当简单常见的做法了。

同步适配器是复杂的猛兽，关于创建它们的完整教程超出了本书的范围。我们的目标是了解它们如何才能工作，这样才能根据记分卡准确地为它们打分。

创建同步适配器是一个多步过程。

- 义链接 ContentProvider 权限和账户类型的同步适配器。
- 义账户类型。
- 定义 ContentProvider。
- 定义 AccountAuthenticator 服务。
- 定义 SyncAdapter 服务。
- 实现 SyncAdapterService 和 SyncAdapter。
- 实现 ContentProvider。
- 实现 AccountAuthenticatorService 和 AccountAuthenticator。

最后两步：实现 ContentProvider 和 AccountAuthenticator 完全超出了定期任务管

非卖品，仅供非商业用途或交流学习使用

理的范围。只要在网络上搜索这两个主题，就能找到各种各样的资源。

注意

ContentProvider 是 Android 中最强大最有趣的抽象之一。ContentProvider 是一种将本地和远程数据之间的差异模糊化的方式，为本地数据集提供了一种 RESTful API。它们还使 URI 通过支持一种特别优雅的模型-视图-控制器（如体系结构），来表示它提供的数据。

Mednieks、Dornin、Meike 和 Pan 在《Enterprise Android》中详细讨论了这个体系结构。

7.6.1 定义同步适配器

创建同步适配器的第一步是编写定义它的特殊 XML 资源。资源定义了本地数据集和远程账户之间的连接。清单 7.6 是一个相当典型的同步适配器定义示例。

清单 7.6 同步适配器的定义

```
<?xml version "1.0" encoding "utf-8"?>
<sync-adapter
  xmlns:android "http://schemas.android.com/apk/res/android"
  android:accountType "@string/account_type"
  android:contentAuthority "net.callmeike.android.simplesync"
  android:userVisible "false"
  android:supportsUploading "true"
  android:allowParallelSyncs "false"
  android:isAlwaysSyncable "true" />
```

前两个属性 `android:accountType` 和 `android:contentAuthority` 是重要的部分。它们创建了账户组和本地数据集之间的关系。

内容权限是本地 ContentProvider 主张权限的空间，它表示数据集。

账户类型类似于内容权限，它们的工作方式是类似的。它是一个字符串，因为一些本地服务要求成为验证者。使用由内容提供者的权限所开始的 URI 数据请求指向由系统确定拥有该权限的 ContentProvider，因此尝试使用由某个 Service 声明的账户类型会被系统指向该 Service 以进行身份验证。

到目前为止，创建、管理和验证账户是创建同步适配器最复杂的部分。由于同步

适配器是内容提供者和账户类型之间的链接，因此账户类型对每个同步适配器来说都是必需的。

然而，账户不是现在讨论的重点。清单 7.6 中使用的账户非常简单。清单中的应用程序本身会声明账户类型的权限，并创建一个这种类型的账户。这个账户不需要身份验证，并且会设置应用程序的账户页是不可见的（像大多数账户那样）。这个同步适配器会用不受信任的远程同步本地数据。

和同步适配器一样，账户类型是在特殊 XML 文件中定义的。清单 7.7 是最小的账户类型定义。

清单 7.7　账户类型定义

```
<?xml version "1.0" encoding "utf-8"?>
<account-authenticator
  xmlns:android "http://schemas.android.com/apk/res/android"
  android:accountType "@string/account_type" />
```

下一步是定义 ContentProvider，即设置链接到账户类型的数据集权限。ContentProvider 是在应用程序配置文件中定义的，如清单 7.8 所示。

清单 7.8　内容提供者的定义

```
<?xml version "1.0" encoding "utf-8"?>
<manifest
  xmlns:android "http://schemas.android.com/apk/res/android"
  package "net.callmeike.android.simplesync" >

  <!-- ... -->

  <application
    android:name ".SyncApp"
    android:allowBackup "true"
    android:icon "@drawable/ic_launcher"
    android:label "@string/app_name"
    android:theme "@style/AppTheme" >

    <!-- ... -->

    <provider
      android:name ".data.SyncProvider"
```

```
        android:authorities "net.callmeike.android.simplesync"
        android:exported "false" />
    </application>

</manifest>
```

要注意，可能令人惊讶的是，清单 7.8 中定义的提供者没有显式导出（android:exported=false）。这意味着对内容提供者的访问由 Android 系统保护。大多数外部应用程序不能访问它。除了 SimpleSync 本身，只有一种应用程序可以访问 SyncProvider，即系统应用程序。如果系统不可信，会有更严重的问题。

注意

不幸的是，提供者属性 android:authorities 的参数不能是一个引用。这有点烦人，因为它意味着权限名必须在几个地方定义，并且这些定义必须匹配。注意编码器。

同步适配器现在已经定义完成了。清单 7.5 显示了 ContentProvider 账户类型的链接，清单 7.7 显示了账户类型的定义，清单 7.9 显示了 ContentProvider 的定义。

定义同步适配器的最后一步是使它在配置文件中可见。由于它在某一刻会变得明显，因此这是在 Service 声明中完成的。清单 7.9 显示了 Service 声明。

清单 7.9 同步适配器 Service

```
<service
  android:name ".SyncService"
  android:exported "false">

  <!-- ... -->

  <meta-data
    android:name "android.content.SyncAdapter"
    android:resource "@xml/sync"/>
</service>
```

当 SimpleSync 应用程序安装在设备上时，Android 系统就会解析配置文件。当对其 Service 的声明进行解码时，它就会发现具有特殊名称 android.content.SyncAdapter 的元数据，并将在其 android:resource 属性中命名的资源识别出来作为同步适配器的

定义。它在同步适配器的系统级数据库中记录定义并完成安装。

7.6.2　同步适配器的工作原理

在继续讨论同步适配器的实现之前，探讨它们的工作原理会很有启发性。

现在假设应用程序 SimpleSync 包含一个已经完全实现的同步适配器，并且它已经安装在 Android 设备中，是时候让同步适配器运行了。图 7.1 演示了 Android 系统调用其同步适配器的过程。

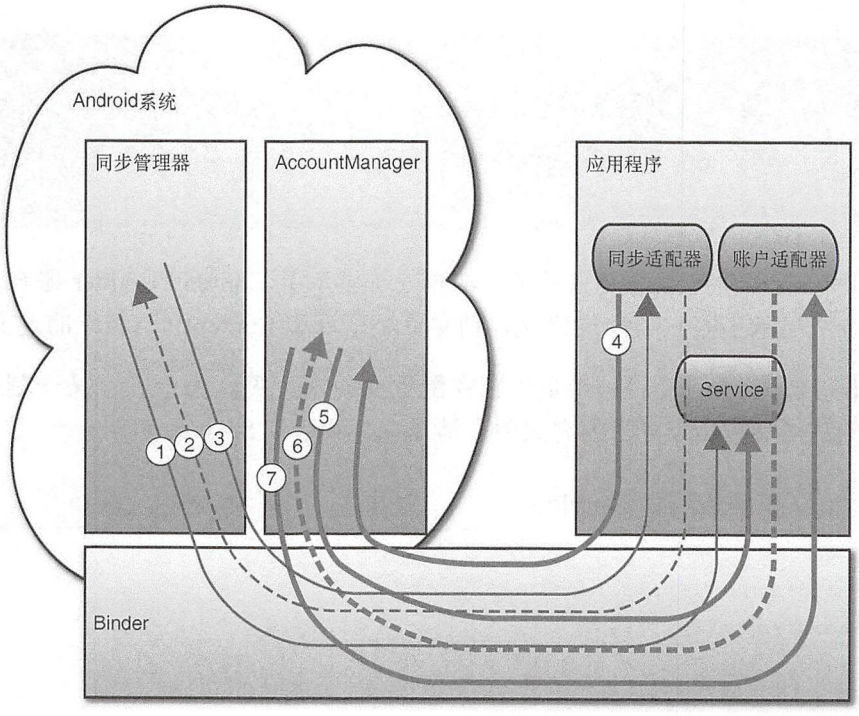


图 7.1　运行一个同步适配器

当 Android SyncManager 服务确定同步适配器需要运行时，这个过程就开始了（见图 7.1 中的箭头 1）。SyncManager（和 AlarmManager）是系统应用程序 ContentService 的组件。它是一个单独的进程，在自己的进程空间中运行。和 AlarmManager 一样，它使用 Binder 建立到应用程序进程间的连接，这个应用程序与特殊的同步适配器相关。和平常一样，如果目标应用程序尚未运行以设置 IPC 连接，重启它就会产生副作用。

第 6 章描述了连接过程。回想一下，要连接到目标应用程序，ContentService 就要查找带有动作 `android.content.SyncAdapter` 的 intent 过滤器的 Service。SimpleSync 应用程序有这种服务。该 Service 是一个工厂，它创建同步适配器对象并将其 IPC 连接返回 SyncManager 中（见图 7.1 中的箭头 2）。SyncManager 现在可以对在目标应用程序上运行的同步适配器进行 IPC 调用。

由于账户类型与配置文件中的声明相关，因此 SyncManager 已经知道要使用哪个账户了，因为它已经准备进行启动同步适配器的调用。当它操作这个调用时（见图 7.1 中的箭头 3），就会将这个账户传递给同步适配器。

回想一下，同步适配器是账户和数据集之间的连接。SyncManager 只为现有的账户运行同步适配器。如果没有与同步适配器相关联的账户类型，那么 SyncManager 永远都不会运行。为了调度同步适配器的定时运行，至少必须存在一个与同步适配器相关联的账户类型。

当一个典型的同步适配器启动时，它通常会验证传递给它的账户。它用几乎与 SyncManager 连接到 SimpleSync 过程相反的 IPC 过程做到这一点。这次目标应用程序对 AccountManager 进行 IPC 调用，而不是将系统服务连接到目标应用程序（见图 7.1 中的箭头 4）上。现在有两个 IPC 连接打开，但这绝不是结束。

由于目标应用程序已经在其配置文件中声明，它负责验证刚刚请求要进行身份验证的类型账户，因此 AccountManager 创建第三个连接以返回目标应用程序的 IPC 连接。再次声明，它做到这一点的方法是，查找注册为账户类型所有者的应用程序，并在其中查找一个具有特定动作 `android.accounts.AccountAuthenticator` 的 intent 过滤器的 Service（见图 7.1 中的箭头 5）。它可能是任何应用程序。但在这个清单中，通常情况下，定义适配器的应用程序也是管理账户类型的应用程序。Service 是账户管理者和其 IPC 连接的工厂（见图 7.1 中的箭头 6）。

AccountManager 服务现在请求账户鉴定者通过 IPC 连接验证账户（见图 7.1 中的箭头 7）。鉴定者对这个过程有完全控制权。在最简单的情况下，账户已经在过去的某个时间验证过，并且最多请求刷新 OAuth 令牌。然而，这个过程也可以很复杂。它可能需要登录到远程网站，读取一个指纹传感器或者转换符号。

最后，至少是在快乐的情况下，鉴定者验证了账户，并告诉账户管理者一切正常。账户管理者告诉同步适配器账户正常，同步适配器运行，在认证账户和内容提供者之间同步数据。

7.6.3　实现同步适配器

从上面的讨论中可以很容易地推断出，需要在 SimpleSync 服务定义中添加其他内容。SimpleSync 应用程序需要为同步适配器本身和（同步适配器相关账户类型的）账户鉴定者充当工厂的 Service。一个 Service 足够了。清单 7.10 显示了从清单 7.9 开始的 Service 的完整定义。

清单 7.10　完整的同步适配器 Service

```
<service
  android:name ".SyncService"
  android:exported "false">
  <intent-filter>
    <action android:name "android.accounts.AccountAuthenticator"/>
    <action android:name "android.content.SyncAdapter"/>
  </intent-filter>

  <meta-data
    android:name "android.accounts.AccountAuthenticator"
    android:resource "@xml/account"/>

  <meta-data
    android:name "android.content.SyncAdapter"
    android:resource "@xml/sync"/>
</service>
```

除了显示同步适配器元数据的定义之外，Service 现在还显示一个账户鉴定者元数据的定义，并将它连接到清单 7.7 所定义的账户类型中。它还定义了一个 intent 过滤器，允许 SyncManager 和 AccountManager 为各自的 IPC 连接查找服务。

正如在同步适配器进程的描述中所指出的，Service 只是同步适配器和账户鉴定者的工厂，它不是两者的实现。根本没有理由认为一个 Service 不可以成为两者的工厂。实现了清单 7.10 中所定义服务的 SyncService 就是这么做的，如清单 7.11 所示。

清单 7.11　同步适配器服务实现

```
public class SyncService extends Service {
  private static final String ACTION_BIND_SYNC = "android.content.SyncAdapter";
```

```
  private SyncAdapter syncAdapter;

  @Override
  public void onCreate() {
    super.onCreate();
    syncAdapter = new SyncAdapter(getApplication(), true);
  }

  @Override
  public IBinder onBind(Intent intent) {
    switch (intent.getAction()) {
      case ACTION_BIND_SYNC:
        return syncAdapter.getSyncAdapterBinder();
      default:
        return null;
    }
  }
}
```

清单 7.11 中的代码创建了 SyncAdapter 的单例实例，并将它返回给任何 onBind 调用，其中 Intent 动作是 android.content.SyncAdapter。如前所述，这是 SyncManager 用于连接 SyncAdapter 的 Intent。

这里我们使用的捷径是，Service 为每个其他请求返回 null。这包含 AccountManager 请求一个账户鉴定者。SimpleSync 没有账户鉴定者，并且永远无法验证账户。然而，如果同步适配器从来不请求账户验证，那么它就能很好地工作。

要注意的是，即使服务返回 null 作为对请求鉴定者的响应，也必须定义一个连接到服务的账户类型，并且该服务必须拥有动作 android.content.SyncAdapter 的 intent 过滤器。除非满足所有这些条件，否则 Android 框架将不会运行同步适配器。

这些都就位后，剩下的就是实现同步适配器了。清单 7.12 就是 SimpleSync 应用程序的适配器。

清单 7.12 同步适配器实现

```
public class SyncAdapter extends AbstractThreadedSyncAdapter {
  private static final String TAG = "SYNC";
```

```
    public SyncAdapter(Context ctxt, boolean autoInitialize) {
      super(ctxt, autoInitialize);
    }

    @Override
    public void onPerformSync(
      Account account,
      Bundle extras,
      String authority,
      ContentProviderClient provider,
      SyncResult result)
    {
      SyncStats stats = result.stats;

      String url = AccountManager.get(getContext())
        .getUserData(account, SyncApp.KEY_URI);
      if (null  = url) {
        Log.w(TAG, "Account has no uri: " + account);
        stats.numAuthExceptions++;
        return;
      }

      Context ctxt = getContext();
      Intent intent = new Intent(ctxt, SyncService.class);

      ctxt.startService(intent);
      try { stats.numInserts += new SimpleSync().sync(url, provider); }
      catch (RemoteException e) {
        Log.e(TAG, "sync failed: " + e);
        stats.numParseExceptions++;
      }
      catch (IOException e) {
        Log.e(TAG, "sync failed: " + e);
        stats.numIoExceptions++;
      }
      finally { ctxt.stopService(intent); }
    }
  }
```

清单 7.12 是 SyncService 创建的同步适配器实现(见清单 7.10),并且系统 SyncManager 通过 IPC 与它连接。同步适配器必须将 AbstractThreadedSyncAdapter 子类化,并且必须实现方法 onPerformSync——它是调用来启动适配器的。

通常,同步适配器会尽早调用 AccountManager 方法 blockingGetAuthToken 来验证传递给它的账户。然而,在这个大大简化的版本中,代码只是检查是否有连接到这个账户的端点 URI,如果没有,就将其作为验证错误来对待。

那么,账户是从哪里来的呢?其代码如清单 7.13 所示。

清单 7.13 简单的账户创建

```
public class SyncApp extends Application {
  private static final String TAG = "APP";
  public static final String CANONICAL_ACCOUNT = "SimpleSync";
  public static final String KEY_URI = "SyncAdapter.URI";

  private AccountManager mgr;
  private String acctType;

  @Override
  public void onCreate() {
    super.onCreate();
    Log.d(TAG, "up");
    mgr = AccountManager.get(this);
    acctType = getString(R.string.account_type);

    if (getAccount() == null) { createAccount(); }
  }

  public Account getAccount() {
    Account[] accounts = mgr.getAccountsByType(acctType);
    int accountNum = accounts.length;
    if (accountNum > 1) { Log.w(TAG, "Unexpected accounts: " + accountNum); }
    return (accountNum < 1) ? null : accounts[0];
  }

  private void createAccount() {
    Account account = new Account(CANONICAL_ACCOUNT, acctType);
```

```
        mgr.addAccountExplicitly(account, null, null);
        mgr.setUserData(account, KEY_URI, getString(R.string.server_uri));

        ContentResolver.setIsSyncable(account, SyncContract.AUTHORITY, 1);
        ContentResolver.setSyncAutomatically(account, SyncContract.AUTHORITY, true);
        ContentResolver.addPeriodicSync(
          account,
          SyncContract.AUTHORITY,
          new Bundle(),
          getResources().getInteger(R.integer.sync_interval_secs));

        Log.d(TAG, "create account: " + account);
      }
    }
```

清单 7.13 中的代码来自 SyncApp 类。清单 7.8 将它注册为 SimpleSync 的应用程序对象。onCreate 方法中的代码确保如果应用程序可以运行，那么就一定存在一种正确的账户类型。

在清单 7.13 中重要的方法是 createAccount 方法，它有 3 个基本要素。

首先，它在顶部创建账户。与账户相关的名称被硬编码为 CANONICAL_ACCOUNT。支持真实账户的应用程序可能会从用户输入中获取名称。

其次，代码将远程 URI 保存为与账户相关的用户数据。在这种情况下，URI 是从应用程序的资源中获取的。它与关键字 KEY_URI 相关，即在清单 7.12 中同步适配器用于恢复它的关键字。虽然这个清单非常简单，但应用程序可以与任意复杂的 URI 相关联，它在账户用户数据中包含账户名、验证令牌和其他必要内容。

最后，代码通过使用 3 个 ContentResolver 方法让它成为一个定时任务。它将在整数资源 R.integer.sync_interval_secs 指定的时间间隔中运行。

还有一个小细节。正如人们所料，干预账户是一种特权行为。让 SyncApp.createAccount 工作需要清单 7.14 所示的 3 种权限。

清单 7.14　内容提供者定义

```
<?xml version "1.0" encoding "utf-8"?>
<manifest
```

```
    xmlns:android "http://schemas.android.com/apk/res/android"
    package "net.callmeike.android.simplesync" >

    <uses-permission
      android:name "android.permission.GET_ACCOUNTS"/>
    <uses-permission
      android:name "android.permission.AUTHENTICATE_ACCOUNTS"/>
    <uses-permission
      android:name "android.permission.WRITE_SYNC_SETTINGS"/>

    <!-- ... -->
  </manifest>
```

7.6.4 对同步适配器评分

构建同步适配器需要很多工作。它需要 Service 和 ContentProvider，还需要账户鉴定者，不管按什么标准，这都是一个复杂棘手的项目。它值得我们付出这么多吗？

同步适配器有一些非常重要的优点。同步适配器和 AlarmService 基本任务都是由进程间通信连接的，并且从长期运行的系统应用程序中驱动的。它们定时运行，不管将它们作为组件的应用程序是否在运行。如果调度同步适配器时，其应用程序还没有运行，那么系统就会将应用程序作为副作用来启动。同步适配器调度会在进程边界中存活下去。

更好的是（和必须在设备断电后重新安装且基于 AlarmService 的调度不同），同步适配器在重启时确实是存活的。同步适配器在设备供电时恢复正常的定时执行，而无须对应用程序的一部分进行任何操作。仅凭这一点就可以使它们作为很多应用程序的首选调度系统。

此外，Android 系统在整个设备运行期间都持有代表同步适配器的唤醒锁。同步适配器保证系统在处理时不会断电，并且系统会正确地释放它所代表的唤醒锁，即使它被异常终止。

同步适配器要在一个相当安全的组件环境中运行。它们正确地将前端、UI 处理与使用该数据的本地缓存来同步远程数据的后端工作分开。同步适配器所需对象的生命周期与适配器本身的生命周期是一致的。将对临时数据的引用保存到静态变量中的常规注意事项仍然适用。然而，同步适配器的环境是相对“自由的”。

不幸的是，虽然同步适配器自动支持电源管理，但它们不自动支持进程管理。同步适配器是一个绑定型（而不是启动型）服务，因此不会对其优先级有特别的提高。若要提高同步适配器的优先级，它必须启动（当然，还有停止）一个服务。

一个既是绑定型又是启动型的服务称为混合服务。清单 7.12 演示了启动由 SyncManager 绑定的同一个服务的混合同步适配器。这个超负荷服务现在有 3 种作用，如下所示。

- 它是 SyncManager 获取从 SyncAdapter 到 IPC 连接的工厂。
- 它是 AccountManager 获取从 AccountAuthenticator 到 IPC 连接的工厂。SimpleSync 没有账户验证，并且总是返回一个空鉴定者。
- 它是同步适配器启动的服务，这个适配器通知 Android 系统有重要的后台工作正在进行，并且包含的进程应该特别注意。

有几种调度同步适配器的方法。最常见的是在指定的时间间隔内运行它，如清单 7.13 所示。和基于 AlarmService 的任务一样，同步适配器是由一个具有全局视野的系统应用程序调度的。它们利用应用程序全局进行批处理，并且具有与无线电和电池类似的电量节约。

然而，运行同步适配器的第二种方式是，在设备已经打开 ICP/IP 连接的情况下调度它运行。如果设备已经有打开的连接，同步适配器就会依靠正在使用的无线电。由于无线电是控制电池电量最昂贵的方式，因此这种调度同步适配器的方法非常有吸引力。

同步适配器也可以显式调度。这种功能的一种常见用法是调度适配器作为对所传入推送通知（例如，谷歌云消息）的响应。使用这种方式时，远程服务器会通知应用程序它应该使用推送通知更新自己，而应用程序运行同步适配器以拉动更新。应用程序成为 MVC 模式中的组件。它表现得很像一个远程模式的视图。

最后，正如一个人可以在给定 MVC 模式中所预测的那样，同步适配器还可以充当控制器，将本地发生的更新推送到远程。这个行为是由 Android 中最可爱的“技巧”完成的。

通知观察者由 ContentProvider URI 所表示的数据已经改变的标准方式是，使用 `ContentResolver` 方法 `notifyChange`。在最常见的用法中，该方法有两个参数：表示数据已经改变的 URI，和对引发改变的观察者引用。清单 7.15 是一个典型示例。

清单 7.15 数据集改变通知

```
public Uri insert(@NonNull Uri uri, ContentValues vals) {
  long pk = getDb().insert(TABLE, NULLABLE_COL, vals);

  if (0 > pk) { return null; }

  uri = uri.buildUpon().appendPath(String.valueOf(pk)).build();
  getContext().getContentResolver().notifyChange(uri, null);

  return uri;
}
```

insert 方法将一行插入到与所传递 URI 相对应的数据库中，然后通知所有已注册的监听器数据集已经修改了。注册的监听器将接收到一个回调，并且据此可以采取适当的行为。例如，和在第 4 章中所讨论相似的 CursorLoader 可能是 URI 表示的数据集的视图。它将接收到通知并重新查询。

除了通知所有监听者之外，ContentResolver.notifyChange 还会启动与 URI 权限相关的同步适配器的运行。例如，如果以其权限组件为 net.callmeike.android.simplesync 的 URI 作为 SimpleSync 应用程序的一部分调用，清单 7.15 中的代码就会导致应用程序的同步适配器在短暂的延迟后运行。用通常的方式通知更改会导致相关的同步适配器将更改同步到远程。使用这种方式，同步适配器会实现完整的 MVC 模式。

还有两个问题。第一个问题是，用这种方式运行时，同步适配器会同步相关类型的所有账户。在小的、单一账户的 SimpleSync 应用程序中，这不是问题。但在支持多个账户的复杂应用程序中，登录过程必须指出当前账户，还必须注册一个显式 ContentObserver 以监听当前 URI 的通知。观察者将强制同步适配器运行来传递登录账户。

第二个问题是，传入更改（例如，响应推送通知的同步适配器）肯定会导致数据库的更改。如清单 7.15 所示，这些传入更改会触发传出同步。这可不太好。

解决方案很简单。ContentResolver.notifyChange 的两个参数版本是一个很方便的重载。完整的方法实际上有 3 个参数。第三个参数 syncToNetwork 是一个布尔值，如果为 false，就表示相关的同步适配器不应该运行。清单 7.16 演示了由网络更新引发数据集修改时，为了阻止同步适配器运行而对代码所做的必要修改。

清单 7.16　通知同步适配器数据集更改

```
public Uri insert(@NonNull Uri uri, ContentValues vals) {
  long pk getDb().insert(TABLE, NULLABLE_COL, vals);

  if (0 > pk) { return null; }

  uri uri.buildUpon().appendPath(String.valueOf(pk)).build();
  getContext().getContentResolver().notifyChange(uri, null, !fromNet(url);

  return uri;
}
```

当同步适配器更新数据库时，它就会用追加查询参数的 URI 做到这一点。fromNet 方法检查这个查询参数是否存在，并在它存在时阻止同步适配器运行。

表 7.6 在记分卡中添加了同步适配器。同步适配器在适用时非常吸引人，尽管它们开销很大。

表 7.6　记分卡：同步适配器

框　架	线程安全	生命周期感知	进程优先级感知	电源节约
Timer 和 TimerTask	NA	NA	NA	NA
Looper/Handler	1	0	0	1
基于服务的调度程序	3	2	3	2
AlarmManager/IntentService	4	4	4	3
同步适配器	5	5	4	4

7.7　JobScheduler

JobScheduler 是在 Android API level 21（Lollipop）中引入的。它的主要限制是由于它是由系统服务驱动的，因此不能将其功能移动到早期版本的 Android 中。在写本书时，JobScheduler 在 1/4 以上的 Android 设备中都是可用的。这个数字将来一定会增加。

激发 JobScheduler 设计的高级概念是对异步任务更细粒度的控制。任务不再只是定期的，相反，在任务执行前，每个任务都与谓词集合相关，并且这些谓词必须是真的。JobScheduler 的架构介于同步适配器和 AlarmManager 系统之间：它是由长时间运

行的系统服务驱动的，但比同步适配器更简单灵活。

运行 JobScheduler 任务只需要 3 步：

- 使任务可被 JobScheduler 访问；
- 调度任务；
- 实现任务。

使任务可被JobScheduler访问和将一个规范许可(android.permission.BIND_JOB_SERVICE)添加到服务组件中一样简单这个组件是抽象类 JobService 的子类。清单 7.17 说明了配置文件中的服务声明。

清单 7.17 JobSchedule Service

```
<service
  android:name ".scheduler.SimpleJobService"
  android:permission "android.permission.BIND_JOB_SERVICE" />
```

要注意的是，和同步适配器不同，这个声明不会自动引发任务调度，这个声明只会使 Service 可被 JobScheduler 访问，以防它被运行应用程序动态调度。

7.7.1 调度任务

调度就是 JobScheduler 所做的一切。作为系统级进程，JobScheduler 有关于设备电源、连接和其他应用程序行为的全局信息。它关于任务何时应该运行的决定，要比到目前为止讨论的其他调度程序“精明得多”。除了全局批处理任务外，它还给开发人员提供了几种其他方法，用于减轻定时任务对电源和项目预算的影响。

调度 JobScheduler 任务所需要的很多代码，看起来都很熟悉。清单 7.18 是一个示例。

清单 7.18 调度 JobScheduler 任务

```
public class SimpleJobService extends JobService {

  // ...

  private static final String PARAM_TASK_TYPE = "SimpleJobService.TASK";
```

严禁网络传播本PDF，违者责任自负！

```
private static final int SAMPLE_TASK = -1;

private static final AtomicInteger JOB_ID = new AtomicInteger();

public static void startSampleTask(Context ctxt) {
  JobScheduler js = (JobScheduler)
    ctxt.getSystemService(Context.JOB_SCHEDULER_SERVICE);

  cancelAllSampleTasks(js);

  PersistableBundle extras = new PersistableBundle();
  extras.putInt(PARAM_TASK_TYPE, SAMPLE_TASK);

  Resources rez = ctxt.getResources();
  int intervalSecs = 1000 * rez.getInteger(R.integer.sample_task_interval);

  js.schedule(
    new JobInfo.Builder(
      JOB_ID.getAndIncrement(),
      new ComponentName(ctxt, SimpleJobService.class))
      .setExtras(extras)
      .setBackoffCriteria(intervalSecs / 2, JobInfo.BACKOFF_POLICY_EXPONENTIAL)
      .setPeriodic(intervalSecs)
      .setPersisted(true)
      .setRequiresCharging(true)
      .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)
      .build());
}

public static void cancelAllSampleTasks(Context ctxt) {
  cancelAllSampleTasks(
    (JobScheduler)ctxt.getSystemService(Context.JOB_SCHEDULER_SERVICE));
}

private static void cancelAllSampleTasks(JobScheduler js) {
  List<JobInfo> jobs = js.getAllPendingJobs();
  for (JobInfo job : jobs) {
```

严禁网络传播本PDF，违者责任自负！

```
            PersistableBundle extras = job.getExtras();
            if ((null != extras) && (extras.getInt(PARAM_TASK_TYPE, 0) == SAMPLE_TASK)) {
              js.cancel(job.getId());
            }
          }
        }

        // ...
    }
```

和平常一样，调用 `getSystemService` 会检索系统服务的引用——本例中是 JobService。和 AlarmManager 一样，这个引用实际上是连接到一个瘦封装器中，通过 Binder IPC 连接的。通过使用 `JobInfo` 类的实例调用其 `Scheduler` 方法 JobScheduler 任务被调度。`JobInfo` 到包含系统服务的应用程序实例反过来将作业标识为运行，并将其条件标识为其运行条件。

JobInfo 对象是可变的且可打包的，并且可以使用 builder 模式来构建。它们的参数被很好地文档化了，并且包含调度定时任务所需要的一切内容。此外，还有一些惊喜。

首先，要注意 JobScheduler 在难以或者几乎不可能使用其他定时调度机制的标准中使调度任务变得简单。在清单 7.18 中调度的任务只在设备连接到电源和不计算流量（可能是 Wi-Fi）的网络中运行。JobScheduler 在这些重要条件中的调度能力是最吸引人的特征。

其次，要注意 JobScheduler 实现了后退。如本章开头所述，一个应用程序范围内的后退标准实现和重试是创建自制执行服务的一个常见原因。JobScheduler 实现了两种后退策略：指数和线性。

虽然清单 7.18 没有说明，但 JobScheduler 还支持截止期限。截止期限是运行相关任务后的最大时间，而不管其他条件是否已经满足。截止期限只适用于一次性任务，不能用于定时任务。

最后，要注意 JobScheduler 调度可以是持久的。一个持久任务不仅要处理进程边界，还要维持动力循环。由于带有持久 JobScheduler 任务的应用程序可以在设备接通电源时自动重启，因此调度一个持久性任务等价于接收 `BOOT_COMPLETED` 广播 intent。因此，只有持有 `RECEIVE_BOOT_COMPLETED` 权限的应用程序才可以调度一个持久任务。

严禁网络传播本PDF，违者责任自负！

和平常一样，将调度程序传递给运行任务的任何自定义参数都必须编组到 Bundle 中，并作为“extras”传递。在清单 7.18 中，extras 机制像在 AlarmManager 清单中使用的那样，用于识别一个要运行的特定任务。

JobInfo 对象的构造函数有两个定义要运行任务的参数。第一个参数是一个整数，它是任务的唯一标识符，它的目标与清单 7.3 中创建 PendingIntent 时提供的任务 ID 完全相同。当需要选择一个特殊的任务（例如，取消它）时，这个整数就会唯一地标识它。由代码保证这个 ID 是唯一的。清单 7.19 使用 AtomicInteger。

构造函数的第二个参数是新内容。JobScheduler 任务被动态创建，并且在配置文件中不是静态设置的（同步适配器中也是如此）。`JobInfo` 对象包含 `ComponentName`。组件名完全指定了要运行的实现任务和包含它的应用程序的 Java 类。

如本节开头所述，任务必须是 Service 组件：它必须在配置文件中声明，它必须是 JobService 的子类，它必须是可访问的，如清单 7.17 所示。然而，任何可访问的服务都可以被调度或者随时取消其调度。JobInfo 对象是一组调度条件和要运行任务的服务组件名之间的唯一连接。

7.7.2　运行任务

和图 7.1 一样，图 7.2 演示了运行 JobScheduler 任务的过程。

当 JobScheduler 确定运行特定任务的条件已经满足时，进程就开始了。和 SyncManager 一样，JobScheduler 是单独的系统级应用程序。它启动一个到应用程序的 IPC 连接，这个应用程序包含在 JobInfo 记录中命名的服务组件（见图 7.2 中的箭头 1）。和平常一样，这样可能会启动应用程序的副作用。

要注意的是，和同步适配器的 Service 一样，JobScheduler 任务的 Service 不需要导出。这是一项安全措施。不导出的服务不是公开可见的，不能被其他应用程序使用。然而，JobScheduler 是一个可以连接到服务上的特权应用程序，即使其他应用程序无法连接到服务。

和任何绑定型服务一样，JobScheduler 连接的服务组件是返回请求对象 IPC 连接的工厂（见图 7.2 中的箭头 2）。JobScheduler 期望服务组件（即 `JobService` 子类的一个实例）将 IPC 连接返回到 `JobService` 子类的一个实例中。也就是说，作为对 JobScheduler 连接请求的响应，服务组件给自己返回代理。`JobService` 类既是任务的工厂，也是这些任务的实现。使用代理的 JobScheduler 调度 `JobService` 方法

严禁网络传播本PDF，违者责任自负！

onStart 以运行它（见图 7.2 中的箭头 3）。

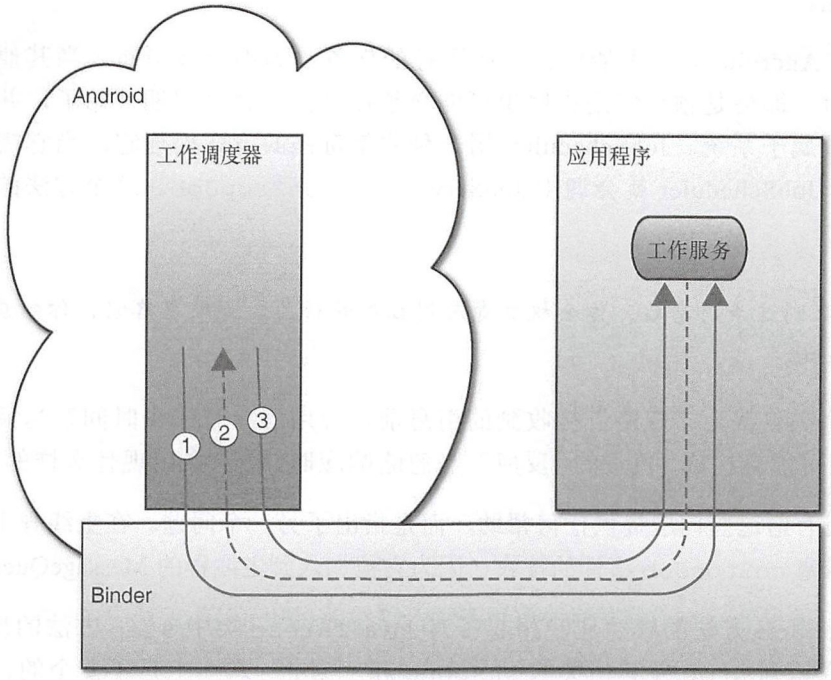


图 7.2 运行 JobScheduler 任务

JobScheduler 最奇怪的特性是，JobService 在主线程上代理 onStartJob 调用返回主线程。当 onStartJob 方法执行时，它就会在主线程上这么做！如果定时调度任务需要时间并且不应该在主线程上运行（这是很常见的情况），那么就必须使用 Looper/Handler 来恢复它。

在远程调用 onStartJob 期间，JobScheduler 持有唤醒锁。然而，当 onStartJob 必须使用 Looper/Handler 异步执行大多数任务时，这显然是不够的。在这种情况下，onStartJob 方法的完成与实际调度任务的完成毫不相关。

为了支持这种情况，onStartJob 方法返回一个布尔值。如果返回值为 true，JobScheduler 就认为作业正在运行，并持有它的唤醒锁直到后续调用 jobFinished 为止。

这种持有唤醒锁的方法要比 IntentService 中使用的技术安全得多。唤醒锁实际上被一种相对安全的系统服务所持有。如果包含调度任务的应用程序由于某种原因而终止，那么不会受到任何影响的系统服务就会释放它。

JobScheduler 在调度任务执行过程中不仅持有唤醒锁，还可以调整优先级，让它

严禁网络传播本PDF，违者责任自负！

的优先级与当前可见的作业几乎相同。在任务运行过程中，驱动调度任务的进程不太可能被杀死。

然而，Android 系统不能保证正在运行的任务永远不会被中断。当其他应用程序需要资源时，即使是预设应用程序也可能必须被终止。整个周期开始了，并且最后的话语权必须属于系统。JobScheduler 用一种非常奇怪的方式处理它。当它需要终止一个任务时，JobScheduler 就会调用 JobService 方法 `onStopJob`。这个方法的文档是这么的说的：

“在接收到这条消息后，你全权负责应用程序的行为。如果忽略它，你的应用程序可能会开始混乱。”

那么，在被认为“忽略”接收到的消息前，应用程序有多少时间？当调用这个方法时，什么可能会构成“负责的”反应？当忽略响应时，可能会出现什么样的“混乱”？

文档没有给这些问题提供任何帮助，而是指出了另一个问题。在主线程上运行的任务本身会阻塞 `onStopJob` 事件的接收（因为它被加入到主线程的 MessageQueue 中）!

`onStopJob` 方法的存在让我想起了对 Java `Thread` 类中 `stop` 方法的反对。Java 的设计者意识到 `stop` 及其相关的 `suspend` 和 `resume` 本质上是不安全的，因而用线程中断协议取代了它们。然而，这个协议要求中断的线程是可协作的，并在中断时停止自己。Android 不仅不能依赖于协作的应用程序，并且它必须期望它们是不可协作的。这些相互矛盾的要求产生了模棱两可的设计。

7.7.3　任务执行

清单 7.19 显示了 JobScheduler 任务的实现。

清单 7.19　JobScheduler 任务的实现

```
public class SimpleJobService extends JobService {
  private final SparseArray<JobParameters> tasks = new SparseArray<>();

  @Inject
  volatile Lazy<SampleTask> sampleTask;

  private Handler taskHandler;
  private int currentTask;
```

严禁网络传播本PDF，违者责任自负！

严禁网络传播本PDF，违者责任自负！


```
@Override
@UiThread
@SuppressWarnings("HandlerLeak")
public void onCreate() {
  super.onCreate();

  HandlerThread thread = new HandlerThread("JobService");
  thread.start();

  DaggerTaskComponent.create().inject(this);
  taskHandler = new Handler(thread.getLooper()) {
    @Override
    public void handleMessage(@NonNull Message msg) {
      switch (msg.what) {
        case SimpleJobService.MSG_START_TASK:
          startTask((JobParameters) msg.obj);
          break;
        default:
          Log.w(TAG, "unexpected message: " + msg.what);
      }
    }
  };
}

@Override
@UiThread
public void onDestroy() {
  taskHandler.getLooper().quitSafely();
  super.onDestroy();
}

@Override
@UiThread
public boolean onStartJob(JobParameters params) {
  enqueueTask(params);
  return true;
}
```

版权所有，严禁传播，违者自负法律责任！

版权所有，严禁传播，违者自负法律责任！


```
@Override
@UiThread
public boolean onStopJob(JobParameters params) {
  cancelTask(params);
  return true;
}

@WorkerThread
void startTask(JobParameters params) {
  int op = dequeueTask(params);
  try {
    switch (op) {
      case SimpleJobService.SAMPLE_TASK:
        if (!Thread.currentThread().isInterrupted()) { sampleTask.get().run(); }
        break;
      default:
        Log.w(TAG, "unexpected op: " + op);
    }
  }

  finally {
    jobFinished(params, false);
  }
}

@UiThread
private void enqueueTask(JobParameters params) {
  synchronized (tasks) {
    tasks.put(params.getJobId(), params);
    taskHandler.obtainMessage(MSG_START_TASK, params).sendToTarget();
  }
}

@UiThread
private void cancelTask(JobParameters params) {
  int jobId = params.getJobId();
  synchronized (tasks) {
```

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
        if (currentTask == jobId) { taskHandler.getLooper().getThread().interrupt(); }
        else {
          JobParameters oParams = tasks.get(jobId);
          if (null == oParams) { return; }
          tasks.delete(jobId);
          taskHandler.removeMessages(MSG_START_TASK, oParams);
        }
      }
    }

    @WorkerThread
    private int dequeueTask(JobParameters params) {
      int jobId = params.getJobId();

      PersistableBundle extras = params.getExtras();
      if (null == extras) {
        Log.w(TAG, "null extras");
        return 0;
      }

      synchronized (tasks) {
        if (null == tasks.get(jobId)) { return 0; }
        tasks.delete(jobId);
        currentTask = jobId;
        Thread.interrupted(); // clear thread interrupts
      }

      return extras.getInt(SimpleJobService.PARAM_TASK_TYPE, 0);
    }
  }
```

这段代码非常长。其中大部分代码都是让调度任务在后台线程上运行所必需的。它和第 5 章中的代码非常相似。

实现只产生了一个后台线程。它无法支持多个同时调度的任务。这是在某些情况下很难接受的限制。一种功能更全面的实现可能必须将任务放在基于自定义服务的调度程序中。

版权所有，严禁传播，违者自负法律责任！

然而，在考虑这些之前，一个开发人员必须考虑对于稍后的执行将任务入队意味着什么，即 JobScheduler 认为应该现在执行的任务，无法保证任何任务的及时执行。显然，如果已经很忙，任何巧妙的代码都没用。如果需要将多个调度程序结合起来，那么就要了解它们在串联时的交互细节，这对创建合理的设计来说至关重要。

调用 `onStopJob` 会将一个消息入队，它包含在后台线程上执行的 Handler 队列的任务参数。它还通过作业 ID 将这些参数存储在稀疏数组中，供以后调用。

当消息在后台线程中处理时，它就会被从队列移除，会识别操作码并且会执行相应的任务。和 AlarmManager/ IntentService 代码类似的实现使用一个服务组件调度多个不同类型的任务。然而，选择这种方式的开发人员必定想创建 JobService 的子类，其中包含大量在清单 7.9 中很明显的样板。

顺便说一句，对于 JobScheduler，没有任何理由不使用 AccountManager。虽然它没有内置支持，但 JobScheduler 任务可以充分利用 AccountManager 服务的灵活性，可以在必要时获取、验证并使用用户账户。事实上，在 Lollipop 早期存在一个谣言：SyncAdapter 会在 JobScheduler 顶部重新实现，以开放对其细粒度调度程序的访问。

除了将请求移动到后台线程的代码外，清单 7.19 中的大部分代码的存在都是因为任务的需要而被停止的。即使是在单线程的情况下，若不考虑停止实际运行任务的复杂性，这也是非常复杂的。

当发生 `onStopJob` 调用时（这是一种一般情况），任务会有两种可能的状态。它会等待执行，或者它正在执行。不管处于哪一种状态，任务都必须停止。

`cancelTask` 方法根据 ID 在任务数组中查找任务。如果任务等待执行，那么原先用于将它入队的 JobParameter 对象就在那里，在 `Handler` 方法 `removeMessage` 中可以使用它删除队列中的任务。

然而，如果任务正在运行，它的任务 ID 就会保存在数据成员 `currentTask` 中。在这种情况下，`cancelTask` 会给运行任务的线程发送一个中断，任务应该遵守这个中断并终止。

7.7.4　对 JobScheduler 评分

JobScheduler 有一些优点。它是到目前为止所检查过的调度程序中最好的调度协议。接口非常简单，用于创建任务调度的生成器使用起来也非常简单。任务是从全局角度调度的，因此可以被智能集群来节约电量和金钱。这种节约能力足以战胜大多数反对意见。

版权所有，严禁传播，违者自负法律责任！

此外，如清单所示，在其他调度程序中标注为很难的事情，很容易在 JobScheduler 中构建。系统安全地管理唤醒锁，并给运行任务的进程足够的优先级，这样它就有足够的工作空间了。

可以通过为一个任务创建不同的调度来增加 JobScheduler 的灵活性。只对清单 7.19 中的代码进行一些明显的扩展，就可以在连接到 Wi-Fi 时每隔 10min 调度一个任务运行，在连接蜂窝数据时每小时调度一个任务，若一整天都没有连接，则通知用户。

JobScheduler 的缺点是其在主线程上运行任务时有奇怪的偏好，以及其粗糙笨拙的作业取消协议。它们会导致很多复杂、易错且鼓噪的代码。

表 7.7 将 JobScheduler 加入表中，JobScheduler 在某些方面表现得非常好。

表 7.7 记分卡：JobScheduler

框　架	线程安全	生命周期感知	进程优先级感知	电量节约
Timer 和 TimerTask	NA	NA	NA	NA
Looper/Handler	1	0	0	1
基于服务的调度程序	3	2	3	2
AlarmManager/IntentService	4	4	4	3
同步适配器	5	5	4	4
JobScheduler	3	5	5	5

7.8 小结

本章讨论了常见的并发范例，定时调度任务和 Android 环境中的几种可能实现。表 7.7 中最后的记分卡显示了每种框架及其主要需求的评级。

在记分卡中完全没有说明的一点是没有单一的银弹。在某些情况下（频繁短暂的任务）Looper/Handler 是正确的解决方案，用同步适配器替换它是十分可笑的。有时，正确的决策是实现基于自定义服务的调度程序，因为它是获得必要控制和灵活性的唯一途径。即使是最近加入的 JobScheduler 也只是一种权衡。

然而，还有其他教训。一个教训是，同步适配器被低估了。尽管有很大的开销和初始成本，但同步适配器是常见的应用程序问题的绝佳解决方案。另一个教训是，了解唤醒锁、进程优先级和硬件的细节，对创建一个健壮令人愉快的应用程序来说至关重要。

版权所有，严禁传播，违者自负法律责任！

第 8 章
并发工具

我们塑造了工具，然后工具又塑造了我们。

马歇尔·麦克卢汉

本章最值得注意的一点是，篇幅非常短。

Java 语言改变了开发者工具链的世界。它的强静态类型使得创建强大的 IDE（例如，Eclipse 和 IDEA）成为可能。这些 IDE 的效用源于这样一个事实，它们可以分析源代码、中间表示和字节码的二进制文件。现代 IDE 理解 Java 代码的深层结构，并且可以执行重构、跳转到定义等复杂操作。

如果没有强类型，就很难可靠地实现这些操作。即使在今天，尽管大多数 Java 开发人员更喜欢结构化 IDE，但经常可以发现动态语言开发人员使用 emacs、Sublime Text 或 Vim。如果没有静态类型，IDE 所提供的功能就不会比简单的轻量级编辑器更多。

然而，Java 在并发方面更像是动态类型语言。即使是在受限的情况下，也很难确定哪些线程可以执行给定的代码。一般来说，这是不可能的。

在这种环境中，只有一部分工具可用于验证并发代码的正确性，而可用于分析和调试有并发错误的应用程序工具更少。

8.1 静态分析

静态分析工具在编译前/后检查代码，并尝试发现问题和不一致。一般来说，这些工具通过应用规则集合来证明代码的语句。由于 Java 的强类型信息可用于这些规则，因此它们是非常有效的。此外，由于它们可以分析代码的结构，因此不需要一定出现极少发生的 bug。

版权所有，严禁传播，违者自负法律责任！

当然，它们不能找到所有问题。由于它们可以找到很多问题，因此在没有通过一个或者多个分析工具运行一个应用程序或者库的情况下释放它，并不比在没有通过拼写检查程序运行它的情况下发送其回复更有意义。

有几种可用的静态分析工具。其中的很多工具主要都是揭露安全问题。很多都要求购买某种许可——很多开发机构都发现这是一种完全合理的投资。在这些常见的以营利为目的的静态分析工具中，具有并发分析功能的工具是 HP Fortify、SonarQube 和 ThreadSafe。

8.1.1 Android Studio

Android Studio 有自己的内置静态分析工具（见图 8.1）——名为 Inspections，它们是在标准 Preference（首选项）对话框窗格中设置的（在 Editor > Inspections 中）。

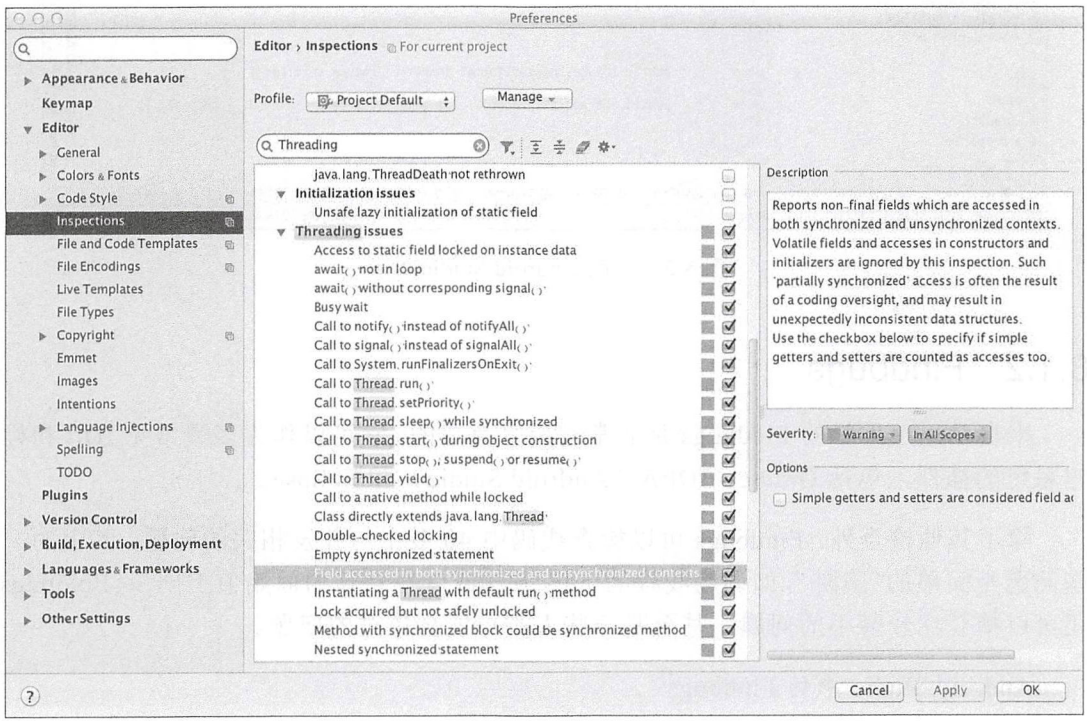
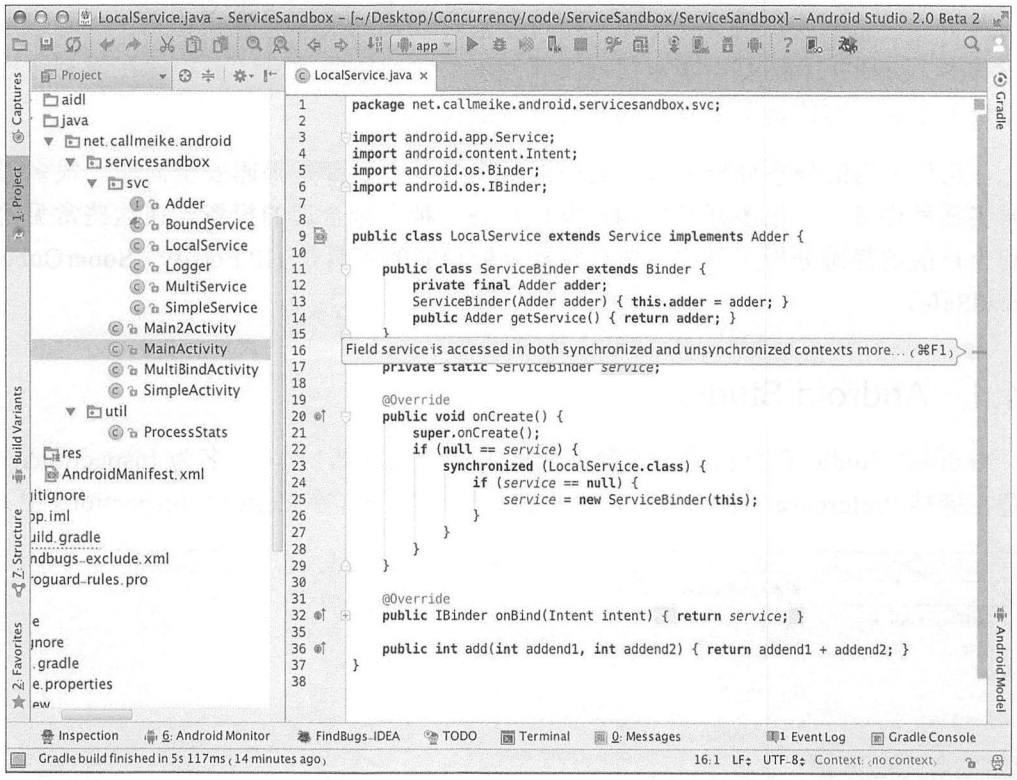


图 8.1 Android Studio 并发检查

Inspections 很好地集成到了 Android Studio 中。编辑器右侧栏中的警告可以检查违规行为，如图 8.2 所示。开发人员所做的可能要比打开所有与并发相关的检查更糟糕，但至少要了解它们生成的每个警告的确切含义。

版权所有，严禁传播，违者自负法律责任！




```
1  package net.callmeike.android.servicesandbox.svc;
2
3  import android.app.Service;
4  import android.content.Intent;
5  import android.os.Binder;
6  import android.os.IBinder;
7
8
9  public class LocalService extends Service implements Adder {
10
11     public class ServiceBinder extends Binder {
12         private final Adder adder;
13         ServiceBinder(Adder adder) { this.adder = adder; }
14         public Adder getService() { return adder; }
15     }
16     private static ServiceBinder service;
17
18
19     @Override
20     public void onCreate() {
21         super.onCreate();
22         if (null == service) {
23             synchronized (LocalService.class) {
24                 if (service == null) {
25                     service = new ServiceBinder(this);
26                 }
27             }
28         }
29     }
30
31     @Override
32     public IBinder onBind(Intent intent) { return service; }
35
36     public int add(int addend1, int addend2) { return addend1 + addend2; }
37 }
38
```

图 8.2　一次 Android Studio 检查

8.1.2　Findbugs

来自马里兰大学的 Findbugs 是非常有用且免费的。它可以作为多数常见 IDE 和构建系统的插件，包括 Gradle、IDEA（Android Studio）和 Eclipse。

除了其他检查外，Findbugs 可以检查代码中 40 多个与并发相关的问题。其中的一些问题是简单的“陷阱”，如调用线程的 run 方法或者在装箱的原始值上同步。Findbugs 还可以捕获比较棘手的问题，甚至是一些人类可能会错过的问题。

Android Studio 中的 Findbugs

在 Android Studio 中配置 Findbugs 的难度稍微高于安装这个插件。使用 Studio 的标准插件安装对话框，浏览插件的存储库，如图 8.3 所示。

插件安装好后，就必须配置它。现在应该在 Preference 对话框的 Other Settings 中有一个条目，如图 8.4 所示。

非卖品，仅供非商业用途或交流学习使用

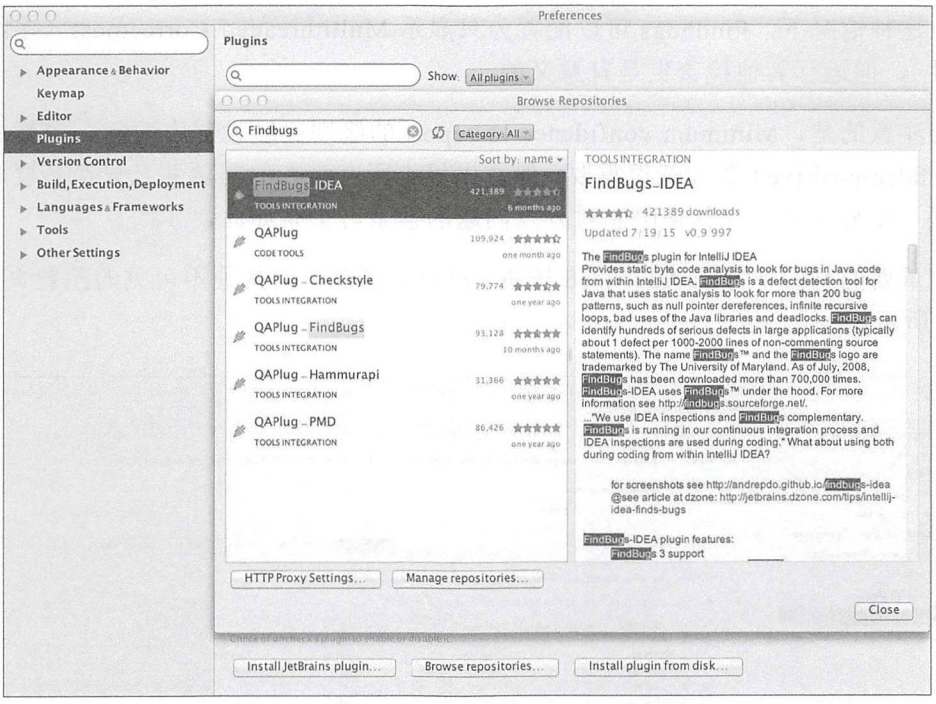


图 8.3 安装 Android Studio Findbugs 插件

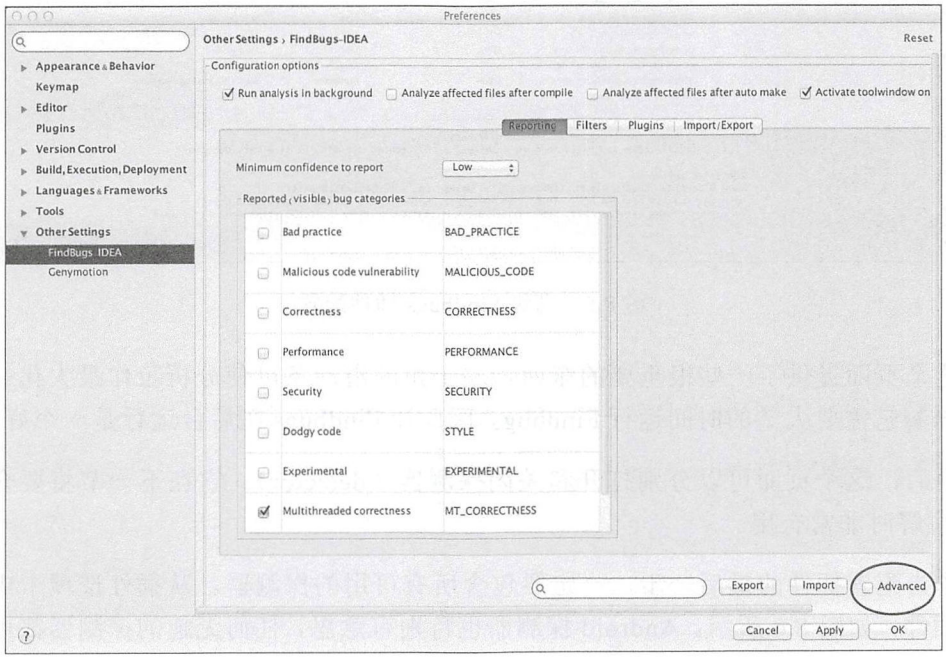


图 8.4 配置 Findbugs 插件

非卖品，仅供非商业用途或交流学习使用

在这种情况下，Findbugs 可以配置为只显示 Multithreaded Correctness（多线程的正确性），但运行其他检查也是有意义的。

要注意的是，Minimum confidence to report 的级别已经设置为 **Low**。虽然保持伪问题（false-positive）数为低很重要，但首先报告所有潜在的问题并且只在必要时进行过滤是一个好主意，尤其是限制对并发问题的搜索时更是如此。

更重要的是右下方的 Advanced 按钮。检查它可以访问其他不可见的配置页面，如图 8.5 所示。

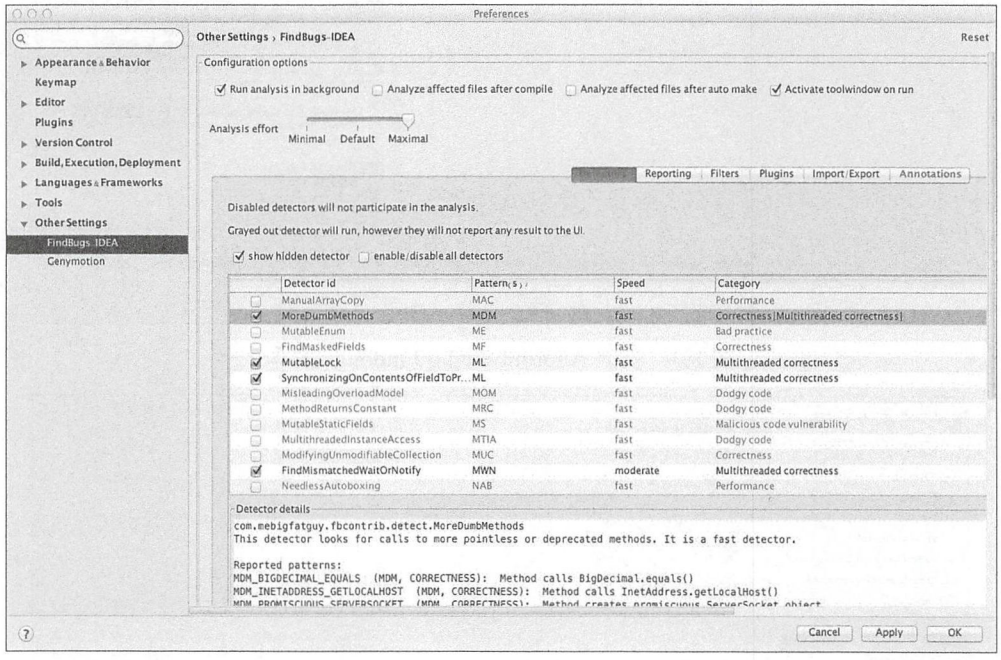


图 8.5　高级 Findbugs 插件配置

这个页面提供了一些很重要的东西。左上角的滑动条可使分析工作最大化。由于这意味着它需要大量的时间运行 Findbugs，因此让 Findbugs 在后台运行是一个好主意。

最后，这个页面可以分别打开和关闭探测器（detector）。它在下一节将要介绍的 JCIP 注解时非常有用。

作为配置插件的最后一步，一定要包含所有可用的探测器。从插件选项卡中检查所有**插件**，如图 8.6 所示。Android 探测器也特别有意思，任何无趣的探测器都可以排除在外。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

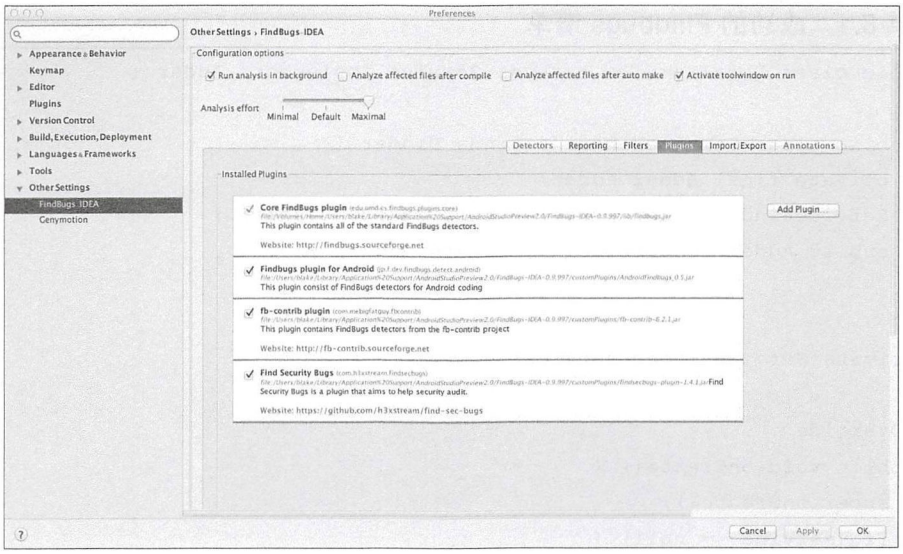


图 8.6 安装探测器

单击 **OK** 按钮完成配置过程。

现在，Findbugs 已经准备运行了。

清单 8.1 显示了一些用于分析的清单代码。显然，它包含一些错误。要用 Findbugs 分析一个项目，首先要选择 Findbugs 并从 IDEA 的 Analyze 菜单中点击 Analyze Project Files。图 8.7 显示了结果。

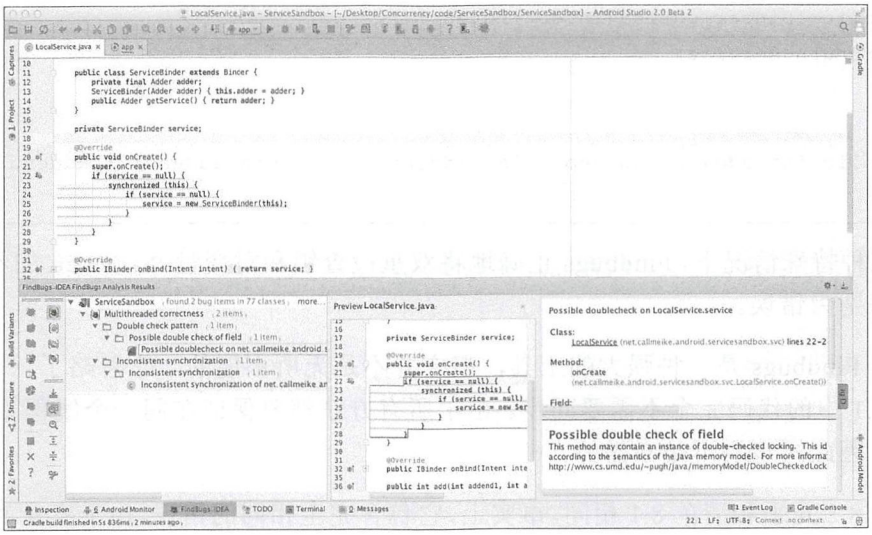


图 8.7 Findbugs 窗格

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

清单 8.1　成功的 Findbugs 清单

```
public class LocalService extends Service implements Adder {

  public class ServiceBinder extends Binder {
    private final Adder adder;
    ServiceBinder(Adder adder) { this.adder = adder; }
    public Adder getService() { return adder; }
  }

  private ServiceBinder service;

  @Override
  public void onCreate() {
    super.onCreate();
    if (service == null) {
      synchronized (this) {
        if (service == null) {
          service = new ServiceBinder(this);
        }
      }
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return service;
  }

  public int add(int addend1, int addend2) { return addend1 + addend2; }
}
```

在这种特殊情况下，Findbugs 正确地将双重检查锁和对变量 `Service` 的不一致同步访问识别为错误。

虽然 Findbugs 是一种强大的工具，但它并不是无所不能的。它忽略了一个事实，即清单 8.1 中的代码完全不需要同步。由于所有方法都要保证在同一个线程（主线程）上运行，因此同步块不是必需的。

清单 8.2 几乎和清单 8.1 相同。事实上，它包含完全相同的错误。由于变量 `Service` 现在是静态的，并且比较的顺序与双重检查锁的第一次检查是相反的，因此 Findbugs

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

错误地给了这段代码一份正常运行的报告书。

清单 8.2 Findbugs 的失败

```
public class LocalService extends Service implements Adder {

  public class ServiceBinder extends Binder {
    private final Adder adder;
    ServiceBinder(Adder adder) { this.adder = adder; }
    public Adder getService() { return adder; }
  }

  private static ServiceBinder service; // static variable

  @Override
  public void onCreate() {
    super.onCreate();
    if (null == service) { // reversed comparison
      synchronized (LocalService.class) {
        if (service == null) {
          service = new ServiceBinder(this);
        }
      }
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return service;
  }

  public int add(int addend1, int addend2) { return addend1 + addend2; }
}
```

注意

你必须用 Findbugs 构建项目以进行分析。

Findbugs 分析已编译的 Java 字节码。为了分析源代码中的变化，该代码段必重新编译。在 IDEA 中，从 Build 菜单中选择 Rebuild Project。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

Gradle 中的 Findbugs

静态分析的一个重要用途是作为持续集成系统的一部分。即使代码库中需要十几分钟运行完整的 Findbugs 分析，运行它作为预承诺方案的一部分也是防止疏忽大意的极好保证。

要做到这一点，Findbugs 必须并入构建系统中。对于 Android 来说，这意味着 Gradle。

从版本 2.10 开始，Gradle 提供基本的 Findbugs 支持。将这种支持集成到 Android 项目中需要一些 Gradle 处理。清单 8.3 的模板可以包含并添加到几乎所有 Android 项目的 `build.gradle` 文件中。它为所有项目构建变体添加了 Findbugs 任务。

清单 8.3　Gradle 中的 Findbugs

```
apply plugin: 'findbugs'

findbugs { toolVersion = "3.0.1" }

afterEvaluate {
  def variants = plugins.hasPlugin('com.android.application')
    ? android.applicationVariants
    : android.libraryVariants

  variants.each { variant ->
    def task = tasks.create("findbugs${variant.name.capitalize()}", FindBugs)

    task.group = 'verification'
    task.description = "Run FindBugs on ${variant.description}."

    def variantCompile = variant.javaCompile

    task.dependsOn(variantCompile)
    tasks.getByName('check').dependsOn(task)

    task.classes = fileTree(variantCompile.destinationDir)
    task.source = variantCompile.source
    task.classpath
      = variantCompile.classpath.plus(project.files(android.bootClasspath))

    task.effort = 'max'
    task.reportLevel = 'low'
```

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！


```
    task.ignoreFailures = false

    task.excludeFilter = file("findbugs-exclude.xml")
    task.reports {
      xml { enabled = false }
      html { enabled = true }
    }
  }
}
```

清单 8.3 中的代码为每个构建变体产生了一个新的 Gradle 任务。每个新任务都为该变体运行 Findbugs。在写本书时，这个清单明确使用 Findbugs 版本 3.0.1。

如清单所示，这段代码会导致这些新任务带有 Findbugs 错误而失败。这在构建作业的上下文中是可取的。想要忽略失败，只要修改布尔值 `ignoreFailures` 即可。

清单 8.3 中的代码也依赖于名为 findbugs-exclude.xml 的 Findbugs 配置文件。Android 项目包含大量自动产生的代码，这些代码会“玩弄”代码标准，从而使得这种排除文件变得几乎必不可少。清单 8.4 给出了这种文件的最小清单。我在这个文件中给出了创建 excludes 文件的更多信息，并在官方在线文档中指出了其他 Findbugs 参数：http://findbugs. sourceforge.net/manual/index.html。

清单 8.4 最小的 Findbugs 过滤器

```
<FindBugsFilter>
  <Match>
    <Class name="~.*R\$.*"/>
  </Match>
  <Match>
    <Class name="~.*Manifest\$.*"/>
  </Match>
</FindBugsFilter>
```

8.2 注解

注解是基本静态分析的扩展。它们是给静态分析器提供提示的额外工具，从而让分析器能够更好地进行分析。

在开发环境中，Android 应用程序代码是根据存根库编译的。由于存根为空且不

非卖品，仅供非商业用途或交流学习使用 严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

包含库方法的实际实现，因此静态分析器会受到严重的约束并且无法看到实际代码。

例如，静态分析工具无法发现 IntentService 的 onHandleIntent 方法和 onCreate 方法在不同的线程上运行。这就给一种常见错误提供了机会：在 onCreate 中初始化字段，然后在没有进行适当同步化的情况下从 onHandleIntent 中访问它。注解提供的提示几乎可以消除这个问题。

8.2.1　JCIR 注解

如前所述，在 Java 中编写并发代码的任何人都应该将《Java 并发编程实践》作为他的案头书。在这本书中，有很多关于并发相关注解的建议。

这些注解，通常简称为 JCIP 注解，它包含在 Java 规范提案（JSR，Java Specification Request）305 中。

- `GuardedBy("guard-variable")`：对如下声明的所有访问都应该在**守护变量**（guard-variable）上同步。
- `@Immutable`：注解类型的实例是可变的。
- `@NotThreadSafe`：一个非线程安全的对象。对象假设为是非线程安全的，因此这个注解大多是文档化的。
- `@ThreadSafe`：这个类型的实例可被多个线程安全地并发使用。

虽然 JSR-305 似乎停滞不前，但包括 Findbugs 在内的一些静态分析工具支持或者部分支持这些注解。写本书时，似乎通过 Findbugs 插件和本地检查（native Inspections）可在 Android Studio 中支持它。Android Studio Findbugs 插件不太可靠。

8.2.2　支持库注解

Android 开发者更感兴趣的是在谷歌的 Android 支持库中定义的 Android 专用注解。其中有几个注解是 Eclipse 和 Android Studio 都支持的。这些与并发相关的注解如下所示。

- `@UIThread`：带注解的方法只在主线程上运行。
- `@MainThread`：它通常和`@UIThread` 相同。根据文档可知，虽然只有一个主线程，但一些应用程序在线程上运行整个替换 UI，而不是这个主线程。文档建议使用`@MainThread` 注释生命周期方法（例如，`onCreate` 和 `onResume`），

非卖品，仅供非商业用途或交流学习使用　　严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

使用@UIThread注释特定于UI的方法（onMeasure和onDraw）。

- @BinderThread：带注解的方法在Binder线程上运行，如第6章中所述。
- @WorkerThread：带注解的方法在上面没有提到的线程中运行。AsyncTask中的doInBackground和IntentService中的onHandleIntent都是使用@WorkerThread注解的方法清单。

本书中的所有代码清单都会在相关时使用这些注解。

这些注解所做的断言要弱于JCIP注解。能让这些注解变得强大的方法是，已经用它们注解开发人员用于构建应用程序的Android框架的存根版本，如清单8.5所示。

清单8.5 使用Android支持的注解

```
public class SimpleIntentService extends IntentService {

  // ...

  @Override
  protected void onHandleIntent(Intent intent) {
    doSomething(); // !!! error
  }

  @MainThread
  private void doSomething() {}
}
```

调用doSomething标记为错误。即使这个清单中的onHandleIntent方法没有被注释，也会在基类IntentService中用@WorkerThread注解注释重写的方法。从由@WorkerThread注解的方法中调用由@MainThread注解的方法显然是错误的。

Gradle lint任务检查Android支持注解，并用在持续集成中运行这些检查。

8.3 断言

静态分析到此为止。某些并发测试必须在运行时完成。

进行运行时并发检查的一种明显的工具是Java的assert语句。从ADB（Android Debug Bridge）命令行输入如下命令就可以在旧版本的Android中使用Java断言：

```
setprop debug.assert 1
```

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

将其包含在文件中，就可以使这个设置持久化：

```
/data/local.prop
```

但不幸的是，从 API level 22 起，Java 断言不再有效。在写本书时，新的 Android 运行时 ART 就已经完全不支持它们了。

虽然这会淘汰一种显而易见的工具，但它并没有消除创建运行时并发检查的可能性。Junit Assert 类中的静态方法是执行类不变量的一种方便的方式。使用它们时，一定要小心地将整个 Junit 套件拖进应用程序中。只使用 junit.framework.Assert，并在必要时使用 minification (ProGuard)消除不需要的类。

如果其他都失败了，一个简单的 if 语句就足够了：

```
if (!Looper.getMainLooper().getThread().equals(Thread.currentThread()) {
    throw new AssertionError(
        "attempt to run this thing from other than the main thread");
}
```

关于断言的重要决定是当它们发生时如何处理它们。这个决定通常是虔诚的，任何开发者或者团队做出的选择可能不会受到这里讨论的影响。然而，大多数开发者团队都同意断言的 3 个目标：

- 将错误报告给能修复它的人
- 不要做任何永久性的损坏
- 不要公布这个问题

也许最常见的选择是只在测试期间启动断言，并在生产中关闭它们。祝我们好运吧！

另一方面，有一个好的故障报告和重启机制的应用程序（例如，Crashlytics、Flurry 等）可能只会在异常时中止。这在某些方面是理想的：问题被报告，应用程序不会在未知状态下继续运行。然而，它通常对用户来说有很明显的缺点。

两者之间存在一个灰色地带，即用某种事件报告器报告错误并尝试恢复。

8.4　结束语和最佳做法

几乎所有 Android 开发人员都必须处理并发代码。要做到这一点很难、很苛刻，并且很容易出错。支持这项工作的工具只到合格线而已。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

可以使用一些最佳做法让代码更健壮，更不易受到奇怪且难以诊断的错误的影响。

- 不可变对象是简单且线程安全的。虽然一些开发人员发现集群很讨厌，但用 final 修饰符明确每个可能的值从而使代码更容易管理。使用 Builder 模式是创建不可变对象的一种好方法。
- 在单线程（不管是什么线程）上运行的代码很容易理解。如果需要多线程，就尽量使它保持住。将它隐藏在框架中，这样大多数开发人员不需要意识到它。
- 作为上一点的推论，尝试让 Java 类和源代码成为单线程的。太大而不适合放入一个编辑器屏幕的源文件，以及包含在不同线程上运行方法的源文件是肯定会发生意外的。
- 如果必须让一个类的方法在多个线程上执行，那么就要注解或者注释这些方法。代码本身并不会表明线程策略。经常听到的借口：“代码就是注释”并不适合多线程代码。
- 非常仔细地检查多线程代码。小心谨慎往往能发现精心计划并良好执行的设计中有微小的疏忽。
- 使用静态分析。将工具上的报告标准设置得尽可能低，并且花时间了解它报告的每个异常。在可能时使用多个工具。注解！
- 将尽可能多的代码移出 Activity（和段）。Activity 是页面的管理器。它不应该包含业务逻辑。
- 从 Android 组件中获取尽可能多的代码。使用一种现代 UI 模式（MVVM，MVP）从 Activity 中获取显示逻辑。将你的 Service 保留为工厂，并通过手工或者使用 IoC（控制反转，Inversion of Control）框架注入实现中。这样测试要容易得多。

为长时间运行的任务创建一些简单清晰的执行策略，然后始终如一地使用它们。在可能的情况下，使用运行时断言以验证它们。让遵循这些架构变得简单并且成为习惯。

祝你好运！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

参考文献

[1] Göetz, Brian, et al. Java Concurrency in Practice[M]. Upper Saddle River, NJ: Addison-Wesley Professional, 2006.

[2] Griffiths, Dawn, David Griffiths. Head First Android Development[M], 1st Edition. Sebastapol, CA: O'Reilly Media, 2015.

[3] Mednieks, Zigurd, Laird Dornin, Blake Meike, Masumi Nakamura. Programming Android[M], 2nd Edition. Sebastapol, CA: O'Reilly Media, 2012.

[4] Moore, Gordon E. Cramming More Components onto Integrated Circuits[J], Electronics, 1965, April 19, 114–117.

[5] Murphy, Mark. The Busy Coder's Guide to Android Development[J], United States CommonsWare, LLC, 2009.

[6] von Neumann, John. "First Draft of a Report on the EDVAC," Contract No. W-670-ORD-4926 Between the United States Army Ordnance Department and the University of Pennsylvania, Moore School of Electrical Engineering, University of Pennsylvania, June 30, 1945.

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

Android并发开发

Android Concurrency

线程与并发在移动系统中和在大型分布式系统中同等重要。本书主要介绍了移动系统开发中线程与并发的基本概念，并描述了如何将它们应用到 Android 框架中。即便你是最有经验的 Android 开发人员，也能从中学到一些技巧。

—Dave Smith, PE, Google Android 开发专家

就开发具有卓越性能和响应能力的软件来说，掌握并发性的相关知识至关重要，对频繁发生突然中断的 Android 系统来说更是如此。为了保证系统的正确性，除了线程安全之外，组件和进程生命周期也是必须明确的。

本书是为有经验的 Android 开发人员编写的，旨在帮助他们学会运用多核处理器和重缓存架构以及内置于 Android 中的主要改进功能。

作为顶级 Android 开发人员和顾问，Blake Meike 揭开了并发编程的神秘面纱，并就从进程间通信到网络通信的所有内容给出了非常实用的解决方案。他还通过引入一个简单而强大的架构框架来解决随时出现的新问题，并且就非常复杂的并发问题给出了建议。

本书将作者渊博的知识、经过验证的模式和语法以及一流的见解结合起来，以期避免破坏性能的错误。对于 Android 开发人员来说，这是不可或缺的资源。

作者简介

G. 布莱克 · 梅克（G. Blake Meike）是一位充满激情的工程师、架构师和“代码诗人”。作为作者、演讲者和导师，他曾教数万人编写真正的 Android 应用程序。他有 20 多年的编程经验（大部分时候是使用 Java 语言），构建过亚马逊可大规模扩展的自动缩放服务系统，也在手机上构建过基于 Android OSS Linux/ 基于 Java 平台的小型系统。他还是很多书籍的合著者，其中包括 O'Reilly 的畅销书《Programming Android》和 Wiley 的《Enterprise Android》。布莱克拥有美国达特茅斯学院的数学和计算机科学学位，并且是 Twitter 大学的创始成员。他住在美国佛罗里达州的奥兰多，就职于 Cyanogen 公司。

通过学习本书，读者将：

- 了解到底什么是并发以及并发进程如何工作；
- 掌握编写并发代码的最佳实践，使这些代码更健壮、不容易出现奇怪且难以诊断的错误；
- 回顾构建 Android 结构的 Java 并发机制；
- 创建一种反映 Android 环境独特特性的并发方法；
- 避免会导致 Android 开发人员陷入麻烦的普遍误解；
- 充分利用 AsyncTask；
- 利用强大且轻量级的 Looper/Handler 框架支持预定的异步任务，并执行多种消息类型；
- 使用 Android Service 组件将业务逻辑从 UI 中分离出来；
- 了解启动和绑定服务之间的区别，并有效地使用它们进行进程内部和进程间通信；
- 执行预定任务，包括需要轮询和明确调度的任务；
- 通过静态分析、注释和断言追踪问题。

Pearson
www.pearson.com

异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

ISBN 978-7-115-48961-6

分类建议：计算机/移动开发/Android
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-48961-6
定价：49.00 元